SPECIFIC TARGETED RESEARCH PROJECT

INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2004-033606

VIsualize all moDel drivEn programming

VIDE

WP 3	**MDD Suitable AO Modelling and Composition Techniques D.3.1** **Suitable Aspect-Oriented Modelling and Composition Techniques in Model-driven Software Development**

Project name:	Visualize all model driven programming
Start date of the project:	01 July 2006
Duration of the project:	30 months
Project coordinator:	Polish - Japanese Institute of Information Technology
Work package Leader:	Fraunhofer FIRST
Due date of deliverable:	30 June 2007
Actual submission date	08 August 2007

Project supported by the European Commission within Sixth Framework Programme

Status	developed / draft / **final**
Document type:	Report
Document acronym:	D3.1
Editor(s)	Jaroslav Svacina, Anis Charfi
Reviewer(s)	*Anis Charfi, Axel Spriestersbach, Joachim Hänsel, Marco Mosconi*
Accepting	Kazimierz Subieta
Location	www.vide-ist.eu
Version	1.0
Dissemination level	**PU**/PP/RE/CO

Abstract

The purpose of work package 3 is to investigate strategies for the integration of aspect oriented composition techniques in model driven development and make recommendations to the other project participants on the most suitable approach for aspect-oriented composition in VIDE. The research goal of Fraunhofer FIRST is to raise AO techniques to a higher abstraction level than programming whereas SAP is seeking techniques for a better modularization and handling of crosscutting concerns in business application models and consequently less complex models that are easy to understand and maintain. This work package addresses the limitations of object-oriented modelling at the PIM level with respect to crosscutting concerns. FIRST presents a proposal for introducing aspect-oriented concepts to the PIM modelling level. Moreover, the semantics of these concepts will be defined and possibilities for implementing aspect-oriented composition using AO-specific model-to-model transformations will be discussed. The work package shows also through examples how the proposed concepts can be used in modelling business applications.

The VIDE consortium:

Polish-Japanese Institute of Information Technology (PJIIT)	**Coordinator**	**Poland**
Rodan Systems S.A.	Partner	Poland
Institute for Information Systems at the German Research Center for Artificial Intelligence	Partner	Germany
Fraunhofer	Partner	Germany
Bournemouth University	Partner	United Kingdom
SOFTEAM	Partner	France
TNM Software GmbH	Partner	Germany
SAP AG	Partner	Germany
ALTEC	Partner	Greece

Executive Summary

The VIDE project aims at developing *"a fully visual toolset to be used both by IT-specialists and individuals with little or no IT-experience, such as specific domain experts, users and testers."*[1]. Therefore VIDE investigates "*visual user interfaces, executable model programming, action- and query-language-semantics, AOP and quality assurance on the platform-independent level, service oriented architecture (especially Web services integration) and business process modelling.*". VIDE is aimed to be embedded in the Model Driven Architecture of the OMG, thus supporting modelling both on a domain-oriented computation-independent layer (CIM), a platform-independent layer (PIM), and generating models on a platform-specific layer (PSM). VIDE is primarily targeting the domain of business application software.

The goal of Work Package 3 in the VIDE project is to investigate integration strategies for adding advanced aspect-oriented software composition in the platform-independent modelling phase of MDD processes. The resulting knowledge allows integrating the aspect-oriented modelling and composition techniques into the VIDE language and architecture. The benefit for the VIDE project will be shown by evaluating the developed concepts and by assessing the used technology.

In this work package we have researched aspect orientation on the PIM level using Customer Relationship Management business scenarios that are provided by SAP. The lack of support in object-oriented modelling techniques for modularizing crosscutting concerns in the provided scenarios raised the need for aspect-oriented techniques while modelling business processes and business applications.

Our research included the evaluation of different existing approaches in the domain of aspect oriented programming by applying them to the relevant phases of Model Driven Development as well as the investigation of existing approaches in the area of aspect-oriented modelling.

Based on the research results a suitable concept for modelling aspect-oriented constructs, such as aspect, advice, and pointcut was developed. To ensure a straightforward integration of these constructs into the VIDE metamodel we have selected the UML Profile extension mechanism.

To allow the VIDE model compiler to deal with the aspect-oriented modelling concepts that we have developed, we present an aspect composition strategy, which is based on model-to-model transformations. The feasibility of the developed concepts and strategies was shown by a proof-of-concept prototype, which consists of UML Profiles for aspect modelling and two transformations respectively for join point matching and aspect weaving at the model level.

Deliverable 3.1 presents the state of the art in aspect oriented composition at the model level and provides an analysis of the chances and risks for the investigated modelling and composition techniques. It also aims at providing the required knowledge for integrating aspect orientation into the context of VIDE.

[1] From the VIDE project summary in the Technical Annex I

Table of Contents

Abstract ..... **- 3 -**

Executive Summary ..... **- 5 -**

Table of Contents ..... **- 6 -**

Abbreviations ..... **- 8 -**

1 Introduction and Overview ..... **- 9 -**
1.1 Challenges ..... - 9 -
1.1.1 Task 3.1: Practical evaluation of AO modelling and composition in MDA ..... - 10 -
1.1.2 Task 3.2: Provision of a knowledge base for AO software composition in MDA processes ..... - 10 -
1.1.3 Task 3.3: The specification of the Aspect-Oriented composition mechanisms to be supported by VIDE ..... - 10 -
1.2 Document Outline ..... - 10 -

2 Background: Aspect-Oriented Software Development and Model Driven Development ..... - 12 -
2.1 Aspect Orientation ..... - 12 -
2.1.1 Introduction ..... - 12 -
2.1.2 Core terms ..... - 12 -
2.1.3 Core concepts ..... - 16 -
2.2 Aspect Orientation at the Model Level ..... - 16 -
2.2.1 Overview ..... - 17 -
2.2.2 Aspect-oriented Modelling ..... - 18 -
2.2.3 Aspect-Oriented Composition in Model Driven Development ..... - 19 -
2.3 Chances and Risks ..... - 21 -
2.3.1 Aspect-oriented concepts at model level ..... - 21 -
2.3.2 Aspect-oriented Modelling ..... - 22 -
2.3.3 Aspect-oriented composition ..... - 22 -
2.3.4 Refactoring of aspect and base models ..... - 23 -

3 Crosscutting Concerns in SAP Business Applications ..... - 25 -
3.1 Example of business application ..... - 25 -
3.1.1 Customer Relationship Management ..... - 25 -
3.1.2 Lead and Opportunity Management ..... - 26 -
3.2 Crosscutting Concerns in the CRM application ..... - 28 -
3.2.1 Consistency checks ..... - 28 -
3.2.2 Partner determination ..... - 31 -
3.3 Benefits of VIDE Aspects in modelling Business Applications ..... - 33 -
3.4 Summary ..... - 35 -

4 Realizing Aspect Oriented Composition in VIDE ..... - 36 -
4.1 Overview ..... - 36 -
4.2 Technological overview ..... - 37 -
4.2.1 Base Technology ..... - 38 -
4.2.2 Aspect Extensions ..... - 38 -
Aspect Modelling ..... - 38 -
Pointcut matching ..... - 38 -
Aspect composition (weaving) ..... - 39 -
4.2.3 Requirements ..... - 39 -
4.2.4 Model Transformations ..... - 39 -
4.3 AOC Architecture ..... - 40 -
4.4 Models and Metamodels ..... - 41 -
4.4.1 Base Models ..... - 41 -
4.4.2 Aspect models ..... - 41 -
4.4.3 AO UML Profile ..... - 42 -

4.4.4 Intermediate model: Join Point Shadow UML Profile .... - 46 -
4.4.5 Result model .... - 47 -
4.5 Model Transformations .... - 47 -
4.5.1 Pointcut Resolving .... - 47 -
4.5.2 Aspect composition .... - 48 -
5 Summary and Conclusions .... - 56 -
5.1 Open issues .... - 57 -
5.2 Outlook .... - 57 -
References .... - 58 -
DISCLAIMER OF SAP AG .... - 62 -

Abbreviations

CIM	Computation Independent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
MDA	Model Driven Architecture
AO	Aspect Orientation
AOP	Aspect-Oriented Programming
AOM	Aspect-Oriented Modelling
AOC	Aspect-Oriented Composition

1 Introduction and Overview

The object-oriented methodology is the most used methodology in software development in general and in model driven development in particular. However, object-oriented decomposition is too limited to efficiently decompose different domains. To address this limitation, aspect orientation provides advanced modularisation and adaptation concepts, especially behaviour adaptation, i.e., inserting and executing encapsulated behaviour at declared interaction points, as well as structural adaptation, i.e., extending already existing structural elements. These additional concepts allow a modularized definition of crosscutting behaviour, non-invasive adaptation of software (components), decapsulation of structure and behaviour, integration of "unprepared" binary components and parameterized specification of behavioural composition.

As a result, MDD can gain a lot from the advanced modularisation concepts provided by aspect orientation. Decreasing the model complexity and favouring an easier reuse and extensibility are among the most important benefits, which allow the modeller to focus on a certain domain and realize functionality without tangling and scattering.

1.1 Challenges

To integrate aspect orientation into model driven development in the VIDE context and show the resulting benefit, three challenges were identified.

The first is choosing the right application scenarios, which are valuable for VIDE partners and users. There are several suitable scenarios for development and production aspects. Examples of development aspects include debugging, testing, performance tuning and monitoring. Additionally production aspects enable the software developer to add functionality without adding more visibility of model element internals, which eases model comprehension, maintenance, and extension.

The second challenge is the integration of aspect oriented concepts into the model driven development process, which makes an exploration of required techniques indispensable. First it is necessary to find out the most suitable way to describe, respectively to model the additional aspect oriented constructs, such as aspect, advice, pointcut, etc. Especially while modelling pointcuts, there are many different variations, which have a significant impact on the resulting power and expressiveness of the mechanism for selecting interaction points in the modelled control flow. After defining a way to model aspect oriented constructs, the composition of the base model and the aspect model has to be investigated. Different approaches from the aspect oriented software development domain are compared and checked for suitability, whereas the main focus is on aspect weaving and instantiation strategies.

At last, the third challenge concerns the development and realisation of aspect composition on the model level in the VIDE context. The non-invasive extension of the VIDE Metamodel using UML Profiles allows the modelling of aspect oriented constructs in VIDE context. The developed concept for aspect oriented model composition is implemented using a set of model-to-model transformations, which perform pointcut matching as well as aspect weaving on the PIM level using different weaving strategies. The realisation shows the feasibility of the developed aspect oriented composition strategy and helps to detect potential technological problems.

These challenges correspond to the following tasks[2].

1.1.1 Task 3.1: Practical evaluation of AO modelling and composition in MDA

A demonstrator (for the sole purpose of the evaluation report) utilizing techniques selected in task 1.3 will be developed, which will show the suitability of the technique, investigate the maturity of its AO modelling approach and spawn hidden risks in composing AO models particularly for data-intense business applications. To this end a metamodel for assessment of most suitable AO approaches will be developed, regarding AO model extensions, aspect weaving level and complexity of MDA transformations. Several parts of a given business application will be analyzed in order to identify composition scenarios crucial for business applications. The most important composition scenarios will be designed and executed using the most reasonable composition technique. The results will be assessed considering the identified factors important to an MDA development process. A report will summarize the experiment's results, discuss the collected data and in particular recommend an AO modelling technique and design for integrating AO composition into the VIDE environment.

1.1.2 Task 3.2: Provision of a knowledge base for AO software composition in MDA processes

By structuring the empirical data of Task 3.1 a standard body of knowledge for best practices of AO modelling and composition techniques in MDA development processes with a focus on the business application domain will be initialized. It addresses the maturity of existing AOP approaches as well as integration issues. The evaluation of this body will take place by dissemination of research results and empirical evaluation by the research community, software companies and tool vendors.

1.1.3 Task 3.3: The specification of the Aspect-Oriented composition mechanisms to be supported by VIDE

Based on the analysis performed and in the cooperation with VIDE language definition activities of WP2, the aspect-oriented composition mechanisms for VIDE will be specified. The specification will cover respective semantics, notation and visual user interface elements.

1.2 Document Outline

After having given an overview of this deliverable in *Chapter 1*, we introduce the core concepts and terms in the domains of Aspect Orientation (AO) and Model Driven Development (MDD) in *Chapter 2*. This part is split into two sections: *Aspect-Oriented Modelling* and *Aspect-Oriented Composition in Model Driven Development*. Then, we present the state of the art in AO and MDD. After that, we provide an analysis of chances and risks in the described approaches.

In *Chapter 3*, some crosscutting concerns in a typical SAP business application are identified and explained. Moreover, the need for aspect-oriented modelling capabilities is motivated and the expected benefits for modelling business applications are discussed.

[2] From the VIDE WP3 description of work in the Technical Annex

Chapter 4 provides a detailed description of the realized aspect-oriented modelling concepts and presents a proof-of-concept demonstrator. Moreover, it introduces the proposed composition strategy, the respective model-to-model transformations, and the chosen technology to implement these. *Chapter 4* also gives examples that illustrate the usage of the realized concepts for modelling the crosscutting concerns identified in *Chapter 3*.

Chapter 5 gives a summary of the proposed approach to aspect-oriented modelling at the PIM level and explains its benefits. Moreover, this chapter discusses open issues and gives an outlook to the future, especially with respect to Deliverable 3.2.

2 Background: Aspect-Oriented Software Development and Model Driven Development

The most pressing problem in software development seems to be complexity. Most target domains and projects get more and more complex. Tackling this complexity during software development needs new techniques and methodologies beside the currently used ones. Both Aspect-Oriented Software Development (AOSD) and Model-Driven-Development (MDD) provide new ways to confine and reduce complexity in creating solution domains and developing software. Both approaches try to solve the complexity problem with different but complementary ideas. So it seems natural to combine these approaches and reap the benefits of both for overcoming complexity in software development.

2.1 Aspect Orientation

This section starts with giving a short motivation for the use of aspect oriented software development in general. After that some frequently used terms of the aspect oriented software development domain will be introduced. This will be followed by a description of weaving techniques employed for compilers of aspect oriented programming languages.

2.1.1 Introduction

The currently mainly used paradigm in software development is object-orientation (OO). After years of object-oriented development, experience has shown that OO is not sufficient enough to modularize certain concerns into the solution domain. Aspect-Oriented Programming emerged as a paradigm which wants to enable a better modularization and encapsulation of crosscutting concerns in software development While OO uses classes as modularization units, AO adds aspects as additional modularization units for crosscutting concerns.

Modularized concerns are composed with association and class-based inheritance in object-orientation. AO extends those with structural composition mechanisms like instance-based inheritance mixins or behavioural composition mechanisms. The behavioural composition mechanism is enabled by well-defined interaction points of aspects and classes (*join points*), declarative specification of interaction points (*pointcuts*), and the interception of the execution to insert behaviour that is defined by an *advice*. Therefore AO programming (AOP) advances the modularization of program behaviour.

In Aspect Oriented Programming (AOP) a huge variety of techniques and concepts is employed to achieve aspect-oriented modularization. Often similar terms denote different concepts. To avoid confusion, the following section introduces the main terms and concepts in aspect-oriented software development.

2.1.2 Core terms

Most of the following definitions are based on [39], [35], and [36].

Separation of Concerns

- Separation of concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements [36].
- Separation of Concerns is an in depth study and realisation of concerns in isolation for the sake of their own consistency (adapted from "On the Role of Scientific Thought" by Dijkstra, [38]).
- Separation of concerns addresses the issue of providing sufficient abstraction for each concern as a modular artefact.

Tyranny of Dominant Decomposition

- The Tyranny of the Dominant Decomposition refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns.
- The Tyranny of the Dominant Decomposition refers to restrictions imposed by this decomposition on the simultaneous use of other decompositions.

Composition

- Composition is bringing together separately created software elements [36].
- Composition is the integration of multiple modular artefacts into a coherent whole.

Weaving

- Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system ([36]).
- Historically this term is used to refer to the composition of aspects with other concerns in the system. (See composition)
- Weaving is the composition of aspects with modules that represent other concerns in the system.

Decomposition

- Decomposition is the breaking down of a larger problem into a set of smaller problems which may be tackled individually.

Modularisation

- Modularization is putting together (or partitioning) artefacts into entities called modules (usually aiming at low coupling and high cohesion).

Module

- A module is an abstraction in the adopted language.

Concern

- A concern is a thing in an engineering process about which one cares [36].

- A concern is a specific requirement or consideration that must be addressed in order to satisfy the overall system goal. [37]
- A concern is an interest which pertains to the system's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders.

Crosscutting Concern

- A crosscutting concern is a concern for which the implementation is scattered throughout the rest of an implementation. [36]
- A crosscutting concern is a concern which cannot be modularly represented within the selected decomposition. Consequently the elements of crosscutting concerns are scattered and tangled within elements of other concerns.
- A crosscutting concern is a concern, which is not modularly represented within the selected decomposition into modules, with as a result the occurrence of crosscutting.

Crosscutting

- Crosscutting is a property of a concern for which the implementation is scattered throughout the rest of an implementation [36].
- Crosscutting is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularise them effectively.
- Crosscutting is a structural relationship between representations of concerns. (Crosscutting is a different concept from scattering and tangling.)
- Crosscutting is the occurrence of scattering and tangling of concerns involving a common module.

Scattering

- Scattering is the occurrence of elements that belong to one concern in modules encapsulating other concerns.
- Scattered concern is a concern which cannot be expressed as a single abstraction within the adopted language (Here the term language may refer to requirement specification, analysis, architecture specification, implementation languages, etc.)
- Scattering is the occurrence of the representation of one concern in multiple modules.

Tangling

- Tangling occurs when the code for the implementation of concerns is intermixed [36].
- Tangling is the occurrence of multiple concerns mixed together in one module.
- Tangled concern is a concern which cannot be expressed as a distinctive abstraction within the adopted language; its definition is not separable from the definition of other concern(s).
- Tangling is the occurrence of the coexistence of representations of multiple concerns in one module.

Aspect

An aspect is a unit for modularizing an otherwise crosscutting concern [39]. It defines structural or behavioural enhancements that are attached to another unit. Most often, an aspect module provides new features, such as pointcut and advice, to define those enhancements.

The aspect module may influence the AO composition in three different ways [20]. An aspect:

- may act as base code to be adapted by other aspects themselves,
- might be specialized into several sub-aspects, and
- may introduce adaptations that cause conflicts.

Join Point

In AOP join points are considered as well-defined "points in the execution of the program" where aspects can interact with other parts of the program. The execution of models requires an adequate definition and considers model elements rather than program elements. Similar to executable program elements, such as statements or expressions, every structural and behaviour model element that appears in the execution of the model can act as a join point. Elements of a structural diagram may represent a join point shadow, specifying where an aspect adaptation can be introduced. A no further restricted *join point shadow* acts as a join point in every model execution. Model elements of behavioural diagrams directly represent specifiable join points. They depict the execution of model elements within a certain scenario. Both kinds of elements are used to formulate an AO adaptation.

A join point model defines all elements that can act as join points during model execution.

Pointcut

A pointcut is a predicate that matches join points [39]. Since join points are points in the execution they comprise static (structure related) and dynamic (execution related) properties. Two kinds of pointcuts can be distinguished: (i) pointcut that select join points by specifying their static properties, i.e., properties of their join point shadows, and (ii) pointcuts that refer to dynamic (runtime) properties, i.e., properties of a specific join point shadow execution. A pointcut is often a member of aspect modules.

Advice

An advice is an artefact that augments or constraints concerns at join points [39]. An advice is the actual behaviour to execute before, after or around a join point [36].

An advice is similar to a method. It defines a list of parameters and contains a block of statements that are executed when the advice is invoked. However, in several AOP languages advices do not have a name and also no return values. An advice is often a member of aspect modules.

Join Point Model

- A Join Point Model (the kind of join points allowed) provides the common frame of reference to enable the definition of the structure of aspects [36].
- A Join Point Model defines the kinds of join points available and how they are accessed and used.

2.1.3 Core concepts

Aspect-oriented composition is generally achieved by combining two model elements. The resulting model element comprises the structure and behaviour of all the elements that were composed. The way in which the structure or behaviour of a particular model element is adapted, i.e., augmented, modified or replaced, is specified by the composition. In general, two specific model compositions can be distinguished: merge of different module structures and the adaptation of a module's behaviour.

Structural Composition

The structure of the resulting elements is produced by merging the structures of two (equivalent) model elements, e.g. two classes or two packages. This symmetric composition allows the introduction of new members and declaration of new module relationships. In contrast to the programming level, also relationships between model elements can be merged as first-class entities.

Behavioural Adaptation

An aspect adapts the behaviour of a model element at a specified join point. This asymmetric composition is specified by a pointcut and binds an advice to a set of join points. The pointcut specifies at which join points the aspect modifies the existing behaviour, and the advice defines the additional behaviour that is executed before, after or around the join point. Behavioural adaptations are in general only navigable from the aspect's side.

In AOP the actual composition is called weaving, which can either be static (at design time) or dynamic (at runtime).

2.2 Aspect Orientation at the Model Level

An important task in Model Driven Development is the creation of precise, complete, platform independent models, which can be transformed into models conforming to different abstraction levels or into code for different platforms. Mainly due to the adoption of *Action Semantics,* UML allows also the definition of executable models.

However, UML and the Action Semantics are based on object-oriented concepts and consequently certain concerns cannot be adequately realized in a modularized way. There is no possibility to encapsulate crosscutting concerns in single design modules. The problems of scattering and tangling arise at the level of UML model constructs. Consequently, the maintenance and evolution of the software and the reusability of existing modules is hindered.

Aspect-Oriented Software Development (AOSD) has proven in the recent years to be suitable for providing technology that allows the encapsulation crosscutting concerns in a modular way. Furthermore, AOSD also provides mechanisms to compose the encapsulated concerns with the software modules they crosscut.

As a result of the additional modularisation concepts the software maintenance and the reusability of software modules is enhanced [15]. Therefore the adoption of Aspect-Oriented concepts has a positive impact on software evolution.

To support the adoption of aspect orientation in industry an adequate tool support is required. The existing tools focus on development environments for aspect-oriented languages, such as AspectJ [16]. Obviously, other kinds of software development approaches, especially Model Driven Development can benefit from aspect orientation.

The following sections describe the incorporation of aspect orientation concepts in Model Driven Development separated into two main parts. The part on aspect-oriented modelling gives an overview about the techniques for modelling and representing the additional AOSD constructs (aspect, advice, etc.). The part on aspect-oriented composition in Model Driven Development provides mechanisms for composing the base and aspect models.

2.2.1 Overview

Some exploratory research in the area of platform independent AOP and the use of AOP in MDA has been performed ([43], [44], [45], [47]). The domains of the aspect and modelling communities are partially overlapping ([46], [17]). Initial research is ongoing, but no conclusions on how to best merge AOP and MDA have been drawn. Some sample tools implement AOP in an MDA setting ([47]). It is likely that the future will merge AOP and MDA into a new paradigm, or will extend the MDA paradigm to crosscutting concern semantics ([48]). Both are powerful methodologies, which solve existing problems in current mainstream paradigms like OO.

As described in the AOP introduction, AOP provides advanced composition concepts. AOP enables a modularized behaviour definition and the use of that behaviour in multiple model elements. With the decapsulation of structure and behaviour, integration of “unprepared” binary components, “a posterior” integration of additional interface hooks and the parameterized specification of behavioural composition AOP helps software developers managing the complexity of software systems.

From the application of AOP to MDA, MDA gains several benefits. The model complexity is decreased and the models become focused on their primary domain. And with parting the models into aspects and models, also the complexity of model transformations can be decreased. All this leads to reduced maintenance expenses. An overview of the exploration is shown in Figure 1.

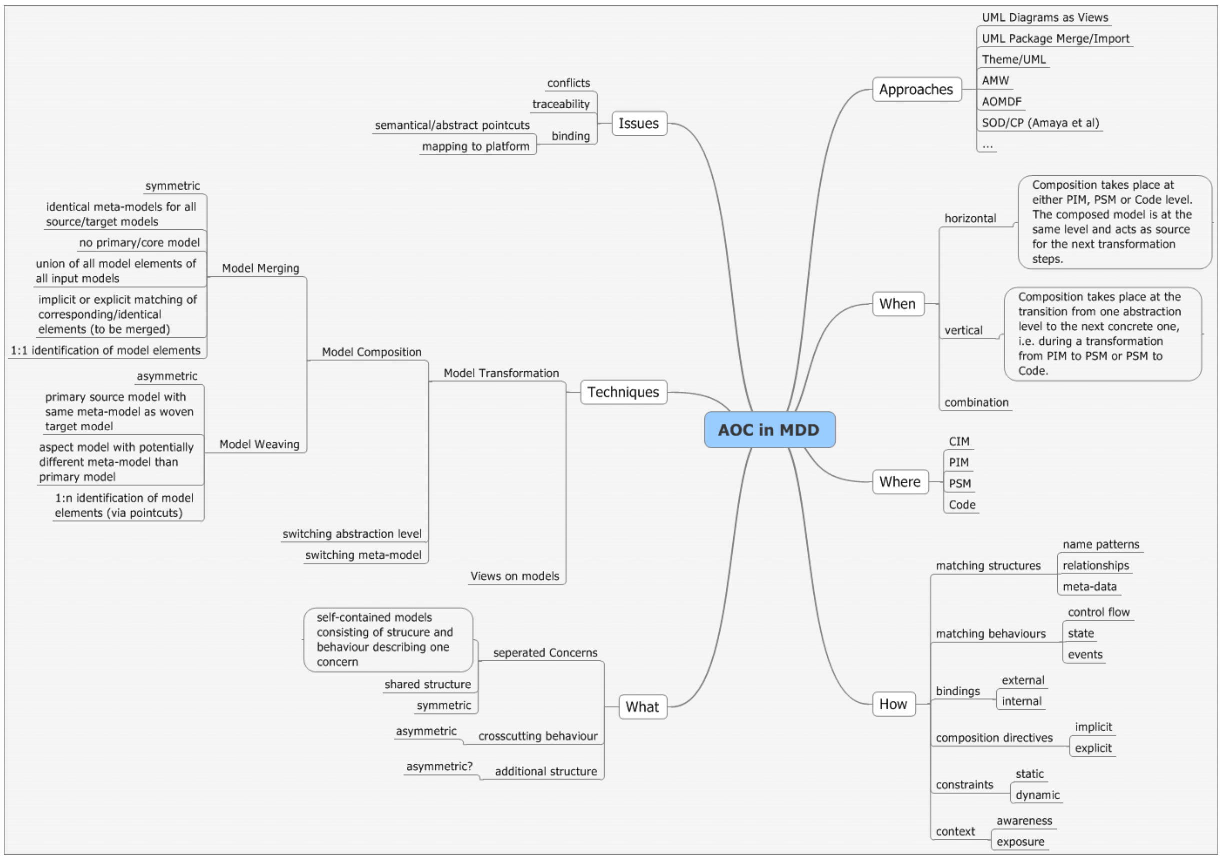
AOC in MDD
Approaches
UML Diagrams as Views
UML Package Merge/Import
Theme/UML
AMW
AOMDF
SOD/CP (Amaya et al)
...
Issues
conflicts
traceability
semantical/abstract pointcuts
binding
mapping to platform
When
horizontal
Composition takes place at either PIM, PSM or Code level. The composed model is at the same level and acts as source for the next transformation steps.
vertical
Composition takes place at the transition from one abstraction level to the next concrete one, i.e. during a transformation from PIM to PSM or PSM to Code.
combination
Where
CIM
PIM
PSM
Code
Techniques
Model Transformation
Model Composition
Model Merging
symmetric
identical meta-models for all source/target models
no primary/core model
union of all model elements of all input models
implicit or explicit matching of corresponding/identical elements (to be merged)
1:1 identification of model elements
Model Weaving
asymmetric
primary source model with same meta-model as woven target model
aspect model with potentially different meta-model than primary model
1:n identification of model elements (via pointcuts)
switching abstraction level
switching meta-model
Views on models
How
matching structures
name patterns
relationships
meta-data
matching behaviours
control flow
state
events
bindings
external
internal
composition directives
implicit
explicit
constraints
static
dynamic
context
awareness
exposure
What
seperated Concerns
self-contained models consisting of strucure and behaviour describing one concern
shared structure
symmetric
crosscutting behaviour
asymmetric
additional structure
asymmetric?

Figure 1: Aspect Orientation in MDD

2.2.2 Aspect-oriented Modelling

Currently aspects are used mostly at the programming language level. With the advent of model driven development and the increasing focus on modelling, several research groups tried to move aspects to the model level. With Aspect-Oriented Modelling (AOM) [49] aspects are integrated in model driven development methodologies.

2.2.2.1 Kinds of Models

There are several approaches to AOM. The first approach is to model a specific programming language aspect framework like AspectJ with UML ([17]). This results in AspectJ typical artefacts and thinking. The second approach is to abstract aspect-oriented development and move it to a conceptually higher level ([18, 19]). After this, the aspect model, composition model, advice model, execution semantics and aspect interactions are expressed in a framework independent way and modelled too ([20]).

2.2.2.2 Notations

Aspects can be modelled with different notations. The most common way is to use a visual notation. This is achieved by extending and customizing UML with UML meta-models and profiles [21]. For most people this is the preferred approach because of its easy tool support and high user acceptance. If UML and the UML extension mechanisms are not flexible enough, aspects can be modelled visually with a custom notation. Most current approaches only use class diagrams for AOM ([22]) and therefore only model structural not behavioural AOP.

2.2.2.3 Modelling Level

As mentioned, aspects are currently used on the programming language level. When lifting them up to the model level, they can be modelled on the computation independent model (CIM), the platform independent model (PIM), or on the platform specific level (PSM). Each level has different constraints on the modelling of aspects and needs different artefacts and probably different visual notations. Beside structural AOP modelling, behavioural AOP modelling is needed especially for the CIM level (for example AOP annotated use cases).

2.2.2.4 Pointcut Languages

Pointcuts connect join points in the target model with aspects. Those connections are crucial in AOM [24, 23]. On the programming language level pointcuts are described with text for example with regular expressions for matching join points [25]. Moving to a model level, pointcuts can also be modelled visually. There do exist several visual pointcut languages, which either directly associate join points with aspects or provide a visual querying language for join points [24, 22], which then connects the visual query description with aspects. Another idea for expressing join points is using colours for each pointcut and aspect combination, underlying join points with colours.

2.2.2.5 Location of Aspects

Aspects and especially pointcuts can be either located in the aspect package, which models a domain, or in a separate package joining two independent domain packages. The later approach enables switching of different pointcut and aspect models and allows developers and modellers to model their domains without knowledge of aspects [26].

2.2.2.6 Crosscutting Concern Visualization

Aspect-oriented programming and modelling is about the encapsulation of crosscutting concerns. A visual modelling framework and visual language probably needs to give visual feedbacks on which join points are adapted by aspects. Otherwise it is hard for the modeller to debug and correctly model specific pointcuts.

2.2.3 Aspect-Oriented Composition in Model Driven Development

Enabling the use of Aspect-Oriented Modelling (AOM) in a model driven setting includes the definition of formal semantics for aspect composition, as the created (aspect) models have to be processed by automated model transformations. Most AOM approaches define concepts for decomposition, but lack corresponding composition semantics [27]. The modelling of aspects and their composition can take place at each abstraction layer in an MDA stack, i.e. CIM, PIM, PSM or Code [28]. Many approaches that deal with aspect model composition propose a composition at the level where aspects are introduced, i.e. mostly at PIM or PSM level. The techniques used for model composition are sometimes called ‘model merging’ and/or ‘model weaving’. In our terminology, *model merging* realises a symmetric composition of models and results in a composed model which constitutes a union of all model elements from the input models.

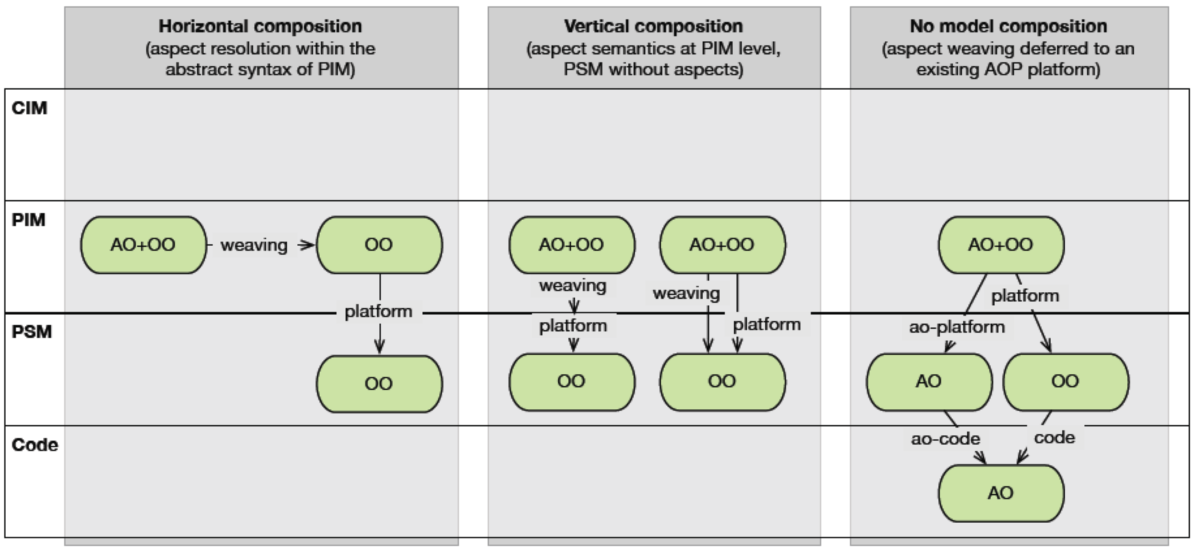

Figure 2: Different kinds of model composition

It is a symmetric composition because of the fact that there is no particular ‘primary’ (or ‘base’ or ‘core’) input model, but all input models are equal. Also, the input models and the merged model are instances of the same meta-model. Elements from different input models that are matched based on an implicit or explicit matching rule (e.g. by name or meta-attributes) get merged as one element in the output model. Following the terminology of AOP, we see *model weaving* as the asymmetric variant of model composition, because it defines one input model as the primary model, which is adapted by one or more aspect models. The meta-models of the resulting model and the primary model (typically not aspect-aware) are the same, while the aspect model can be based on a different (typically aspect-aware) meta-model. Model weaving also introduces quantification, which allows for 1:n matching of model elements and thus weaving of elements of an aspect model into multiple elements of the primary input model.

Conceptually, model merging and model weaving are specializations of model transformation and can therefore be realised through standard model transformation techniques [29]. In contrast to model transformations as used in an MDA context, model composition generally does not switch abstraction levels or meta-models of the involved models.

A completely different approach beside model transformation is the concept of partial views on one common model repository. This approach is found in most UML-Tools, where each diagram depicts only a part of the model. In this case, no explicit composition step is necessary, because the complete and consistent model is always present in the repository. In this approach, the modularization and separation of concerns would become an issue of the modelling tool that would have to integrate the aspect views dynamically.

We identified 4 variables that describe properties of different model composition approaches:

- ***‘Where’*** - Where are aspects defined and/or composed?
  Possible locations are CIMs, PIMs and PSMs as well as the source code [30]. Most AOM approaches fit into PIM or PSM level, because they are extensions of the UML. AOM languages representing concepts of a concrete AOP platform should be

considered platform-specific, because the underlying aspect composition semantics is dependent on this particular platform.

- ***'When'*** - When is the composition performed?
  Composition can be performed in a horizontal or a vertical transformation step (See Figure 2). Horizontal composition means that the composition takes place either at PIM, PSM or Code level. The composed model stays at the same abstraction level and acts as a source for the next transformation steps. A vertical composition takes place at the transition from one abstraction level to the next concrete one, i.e. during a transformation from PIM to PSM or PSM to Code. When model composition can be performed directly at the level where the aspects are modelled or can be delayed to a later point, i.e. a more platform specific level [30, 31].
- ***'What'*** - What gets composed?
  Symmetric approaches allow the definition of modules that are self-contained and independent of each other. These modules constitute models consisting of structure and behaviour describing one concern [32, 33]. On the other hand, in asymmetric approaches it is often crosscutting behaviour that needs to be integrated in one or more elements of other models [31]. The introduction of additional structure to existing model elements is also possible.
- ***'How'*** - How does the model composition work?
  In the first place, model composition is about matching and integrating structures ("static" model elements) and behaviours ("dynamic" model elements). These are typically identified by name patterns, explicit relationships or meta-data and in the case of behaviours based on control flow, state or events. For asymmetric model composition, the bindings of primary model elements to aspects have to be defined. These bindings can be part of the aspect model or outside of the models. Other possible configuration artefacts for model composition can be constraints and composition directives. The former can further restrict identification and matching of elements from different models, the latter define additional rules for the integration of model elements [31, 34].

2.3 Chances and Risks

As described in the previous sections, there are many approaches for the adoption of aspect-oriented concepts to the PIM level. The usage of aspect-oriented modelling und composition techniques comes up with many advantages and also with some problems, which could hinder the integration into the existing model-driven development processes.

The next sections discuss the chances and risks with respect to different domains.

2.3.1 Aspect-oriented concepts at model level

Aspect-orientation provides additional concepts for extended modularisation of both structure and behaviour of crosscutting concerns. These concepts enhance the modularisation, reusability of modules and allow the reduction of complexity, which have a positive impact on the evolvability and maintainability of the software.

Since model-driven development also aims at an improved evolvability and maintainability, consequently the adoption of aspect-oriented concepts into the model-driven development will achieve a good developer and modeller acceptance.

2.3.2 Aspect-oriented Modelling

To enable the usage of aspect-oriented concepts during the modelling phase, new types of model elements are required to represent the additional aspect-oriented constructs, such as aspect, advice, Pointcut, etc.

In terms of the modelling standard, UML provides different ways, to extend the UML metamodel for defining the additional constructs. The different ways can be distilled into two kinds of metamodel extensions:

- Direct changes to the UML metamodel (heavyweight extension)
- UML metamodel extension using UML Profiles (lightweight extension)

2.3.2.1 Extended UML Metamodel

The heavyweight variant of extending the UML metamodel to support aspect-oriented constructs causes direct changes to the UML metamodel. Due to these changes, existing tools and modelling environments have to be modified or generated again. On the other hand, in comparison to the UML Profiles, this variant provides a more flexible extension mechanism.

If this extension mechanism is used to present aspect-oriented constructs, the high effort for integration into existing environments could be a reason for avoiding the usage of the aspect-oriented extension.

2.3.2.2 UML Profile

The UML Profile mechanism provides a lightweight mechanism to present additional types of model elements, such as aspect-oriented constructs. The UML metamodel is not changed during the extension. The UML Profile mechanism provides only an additional extension, which conforms to the original UML metamodel. Therefore many existing tools and modelling platforms are able to deal with an extended metamodel using the UML Profile mechanism.

Because of the described characteristics, in comparison to the changed UML metamodel, the realization of the extension using UML Profiles causes less efforts. So the UML Profile mechanism is predestined for proving the concepts for aspect-oriented modelling as well as for basic integration of aspect-oriented concepts into existing modelling tools.

2.3.3 Aspect-oriented composition

This section discovers the chances and risks with regard to the relevant parts of the aspect-oriented composition. For this purpose the chances and risks for different kinds of aspect composition and for the underlying technologies are analysed.

2.3.3.1 Different kinds of aspect composition

As described in previous sections, different composition techniques can be used for the aspect composition. In *horizontal composition*, the transformation is processed at the PIM level, i.e., the input models and output models are both at the same level.

This kind of transformation allows a nearly independent integration of the aspect composition into existing MDD processes. The transformation handling the aspect composition can be processed before a model compiler transforms object oriented PIM level models into code for example. In this case, no extension of the model compiler is necessary, because the transformations, responsible for aspect composition, produce composed object-oriented models, which can be consumed by the model compiler in a common way.

Since the aspect composition is done at the PIM level in a horizontal way, traditional object-oriented models are processed in the following phases, e.g. transformation to code. From this it follows that on the next abstraction level (e.g. PSM, code) no special support for aspect-oriented concepts is required. So it is possible to adopt the aspect-oriented concepts at PIM level and consequently produce code for every platform that was supported before.

Furthermore, due to the independency of the target platform and the nearly independent integration into the model-driven processes, the horizontal aspect composition at PIM level has a good chance, to be established.

In *vertical composition*, the model transformations occur across different model/code levels. Depending on the point where the aspect composition is processed, additional support of aspect-oriented concepts is required at another model level or at code level. Therefore in some cases the integration of aspect-oriented concepts cannot be done independently without providing support for aspect-orientation at another model level or at the code level.

Anymore due to the composition across more than one level, the aspect composition has to be integrated into existing model transformation tools or into an existing model compiler. This integration causes more effort than an integration of the horizontal aspect composition, because horizontal composition is integrated with just an additional step in the whole process.

Also the integration of new concepts into existing model transformation tools respectively a model compiler is more error-prone in comparison to the independent integration of the horizontal aspect composition into the existing MDD process.

2.3.3.2 Technologies for aspect composition

In the domain of model-driven-development there are proven and tested technologies for processing model-to-model or model-to-code transformations. The new transformations required by the aspect composition can be realized using these proven technologies, such as the *Atlas Transformation Language (ATL).*

The usage of such evaluated technologies reduces the technological risk with regard to the integration of aspect composition into existing MDD processes.

2.3.4 Refactoring of aspect and base models

The technique of Refactoring was identified as a means to increase software’s quality and software’s ability to evolve. Refactoring provides a mechanism for removing or at least improving structural weaknesses. It changes the structure of the software in such a way that the behaviour of the system is not changed yet its evolvability is improved. Consequently the refactorings should also be used at model level to improve the evolvability of a modelled software system.

The refactoring of aspect-oriented models causes similar problems like refactoring of aspect-oriented programs. One of the problems, which could be a risk for the acceptance of the aspect-oriented modelling, will be outlined in the sections below.

Pointcuts provide a mechanism for additional referencing of points in the control flow. Regarding to the UML Action Semantics the so-called join points are also model elements. These model elements are not referenced directly by the pointcuts. For selecting these model elements (Pointcut resolving), the pointcut quantifies over their properties, such as join point kind, signature, etc.

However the properties used by the pointcut for selecting certain model elements can be changed by a refactoring (e.g. Rename Refactoring changes the name of a model element), so if a pointcut uses a property of a model element, which is changed by a refactoring, the result of resolving this pointcut after the refactoring can differ from the resolving result before the refactoring. The difference in the selected join points can also cause a changed behaviour, because due to the changed set of join points the bound advice is called at different points.

Since the selected join points are not referenced explicitly using an association, the model validation is not able to detect the changed behaviour. Therefore a refactoring of model elements can cause unintentional changes in the modelled behaviour.

Even if a mechanism for detecting a changed set of join points before and after a refactoring could be provided (e.g. like the PCDiff tool [40]), it is still not possible to decide automatically whether the detected changes in the behaviour are explicitly intended by the pointcut modeller.

Likewise it would be hard for the base model developer to check all impact on the set of join points manually. Typically the modellers roles are separated into base model developer and aspect model developer, therefore even if the modeller would automatically be informed about the impact of the refactoring, he would not be able to decide, if the changed behaviour was intended by the pointcut developer (without communication to the aspect developer)

To deal with the described problem, the following tool support would be necessary:

- Detection of the refactoring impact on the set of join points
- Assessment of the modified references (modified behaviour intended or not) and detection of broken pointcuts
- Automatic update of broken pointcuts, if possible
- Communication platform between base model developer and aspect model developer in the following cases:
    - Impossible detection of broken pointcuts
    - Automatic update of broken pointcuts impossible

Since refactoring is an elementary technique for improving the evolvability in software development, the absence of suitable refactoring tool support for base and aspect models represents a risk for the establishment of aspect-oriented concepts at model level.

3 Crosscutting Concerns in SAP Business Applications

This section starts by presenting an object-oriented business application from SAP in the context of Customer Relationship Management. After that, some crosscutting concerns are presented such as consistency checks and partner determination. When modelling such concerns with the traditional means of UML several issues arise. These issues can be addressed by using modelling aspects.

3.1 Example of business application

Business applications span various areas and processes in corporations such as Product Life Cycle Management (PLM), Supply Chain Management (SCM) and Customer Relationship Management (CRM). In the following, we will focus on SAP business applications for Customer Relationship Management (CRM) such as *SAP CRM*.

CRM is a management concept, which intends to systematize and improve the relationships between corporations and their customers. It is a customer-oriented corporate strategy that utilises modern information and communication technologies to establish long-term, profitable customer relationships through holistic and individual marketing, sales and service instruments [3].

3.1.1 Customer Relationship Management

CRM software provides a central point to manage all contacts and interactions of a company with its customers. CRM software covers two functional areas:

- Operational CRM: supports the three CRM processes: marketing, sales, and service. These processes reflect the different phases in the “customer buying cycle” [4]. Operational CRM software provides applications and tools for supporting and controlling the different customer interaction points and communication channels. Common core functionalities of such software include contact management, report generation, workflow, and activity management.

- Analytical CRM: stores all relevant data about customer contacts and reactions (e.g. purchase data, billing and payment, returns) in a data warehouse. This data may be combined with other external data before it is analysed using data mining methods or used for answering on-line analytical processing (OLAP) queries.

SAP offers several CRM products such as mySAP CRM, which was recently renamed to SAP CRM [5] [12]. This application supports the entire operational CRM field and provides components and functionalities supporting the three fundamental CRM processes marketing, sales, and service. This application is implemented using object-oriented programming and some important business objects of each process are grouped together below:

- **Marketing**: Lead
- **Sales**: Opportunity, Customer Quote, Sales Contract, Service Contract, Sales Order, and Service Order
- **Service**: Customer Return, Service Request, and Service Confirmation

3.1.2 Lead and Opportunity Management

Figure 3 shows a Sales Scenario example that focuses on sales processes of enterprises selling one or more products. This involves different things, ranging from Opportunity Management to quotations to customers, sales orders and invoice processing. This figure shows also the different user roles that are involved in each step in the sales process.

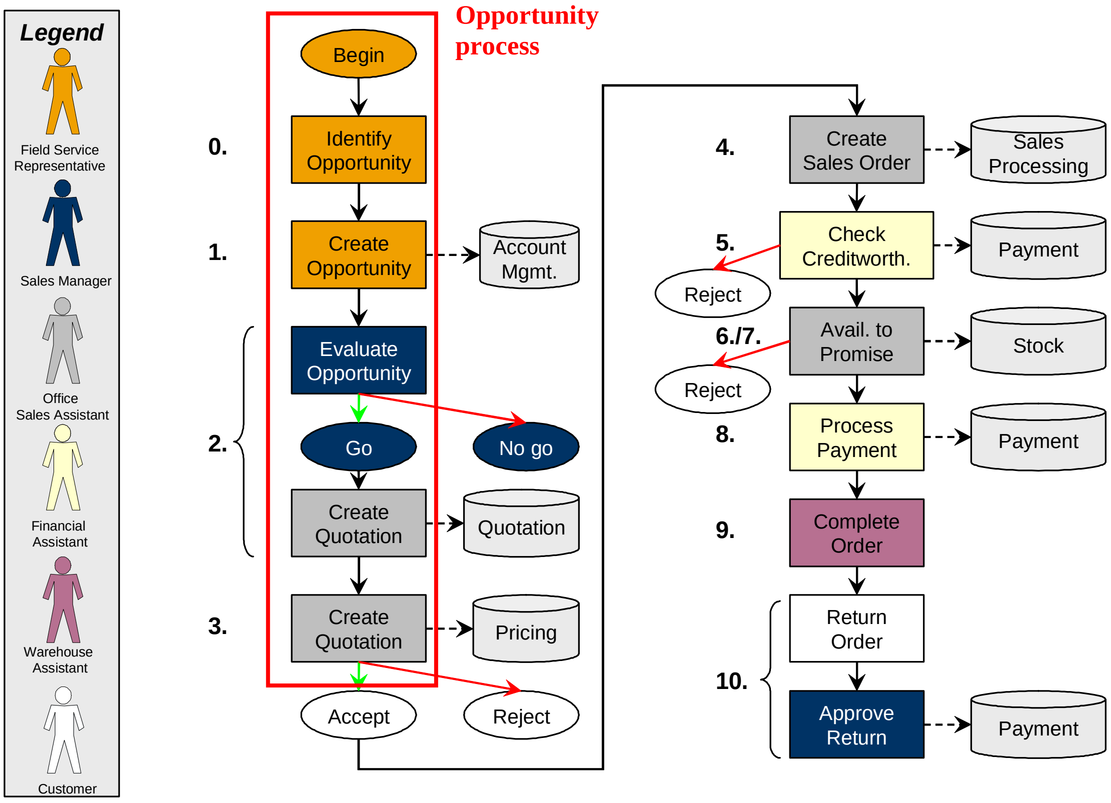

Figure 3: Sales Scenario

In the following, we will focus on pre-sales processes such as lead management and opportunity management. These processes support sales personnel in actively tracking potential selling possibilities.

Lead and Opportunity management provides a structured approach to turning an initial recognition of a selling opportunity (i.e., a potential possibility for selling products to a customer) into a sales contract. In that process, the SAP CRM software guides the sales representative through a multilevel process and generates next steps and activity suggestions on the basis of best-practice sales strategies.

The opportunity management process may start with an anonymous address and, by degrees, track additional prospect attributes such as product interests, discretionary budget amounts, likely competitors, and the success probability. Completeness and consistency checks ensure the correctness of the collected data after each step. The accurately documented process improves reporting capabilities: Sales managers can measure their salesperson productivity, campaign effectiveness and can, for example, determine in which sales phases the most prospects were lost [6,7].

Figure 3 shows also the different steps of the opportunity process. This process starts by the identification and the creation of an opportunity, e.g., after a sales contact at a fair. Then, the opportunity is evaluated and qualified, i.e., feasibility is clarified, information is gathered about the customer, and a selling team is defined. If a go decision is made, a quotation is made and sent to the customer, which either accepts the sales offer or rejects it. After that the opportunity should be closed and the reasons for success or failure should be documented. In the success case, the opportunity becomes a sales order.

In the following, we present in more details some business objects in opportunity management. These objects are shown in Figure 4 and they are discussed briefly below:

- The ***Opportunity*** class uniquely identifies the opportunity and specifies the various involved parties. It holds references to other classes with additional business information and to the documents and activities created during opportunity processing. Some direct attributes of the opportunity class are:
    - *priority*: specifies the priority of the opportunity.
    - *processStatusValidSinceDate*: the date when the opportunity entered the current life cycle phase.

- The ***Party*** class represents individuals or organizations involved with the opportunity. Specialized classes may represent customers, suppliers, or employees. Parties are used within the opportunity to specify the prospect, potential competitors, the responsible sales team, and other internal or external stakeholders. Some attributes of a party are:
    - *partyType*: specifies whether a party is an organization, a business partner, or any specialization of these party types.
    - *partyRole/PartyRoleCategory*: describe the role of a party in an opportunity.

- The ***SalesForecast*** class contains estimations for the anticipated sale that an opportunity represents. it contains various fields such as
    - *expectedRevenueAmount*: the expected amount of the opportunity
    - *probability*: the success probability of the opportunity, expressed in percentage.

- The class ***Item*** represents a product or service which will possibly be sold to the prospect of the opportunity. It contains product information, quantities, and values. An item may be associated with master data product information.

- An opportunity passes through several phases during its lifetime. The class ***SalesCycle*** specifies the sales cycle and the current phase of an opportunity. Other attributes of this class are:
    - *salesCycleCode*: the sales cycle in which the opportunity exists.
    - *phaseProcessingPeriod*: the time period for which an opportunity exists in the current phase.

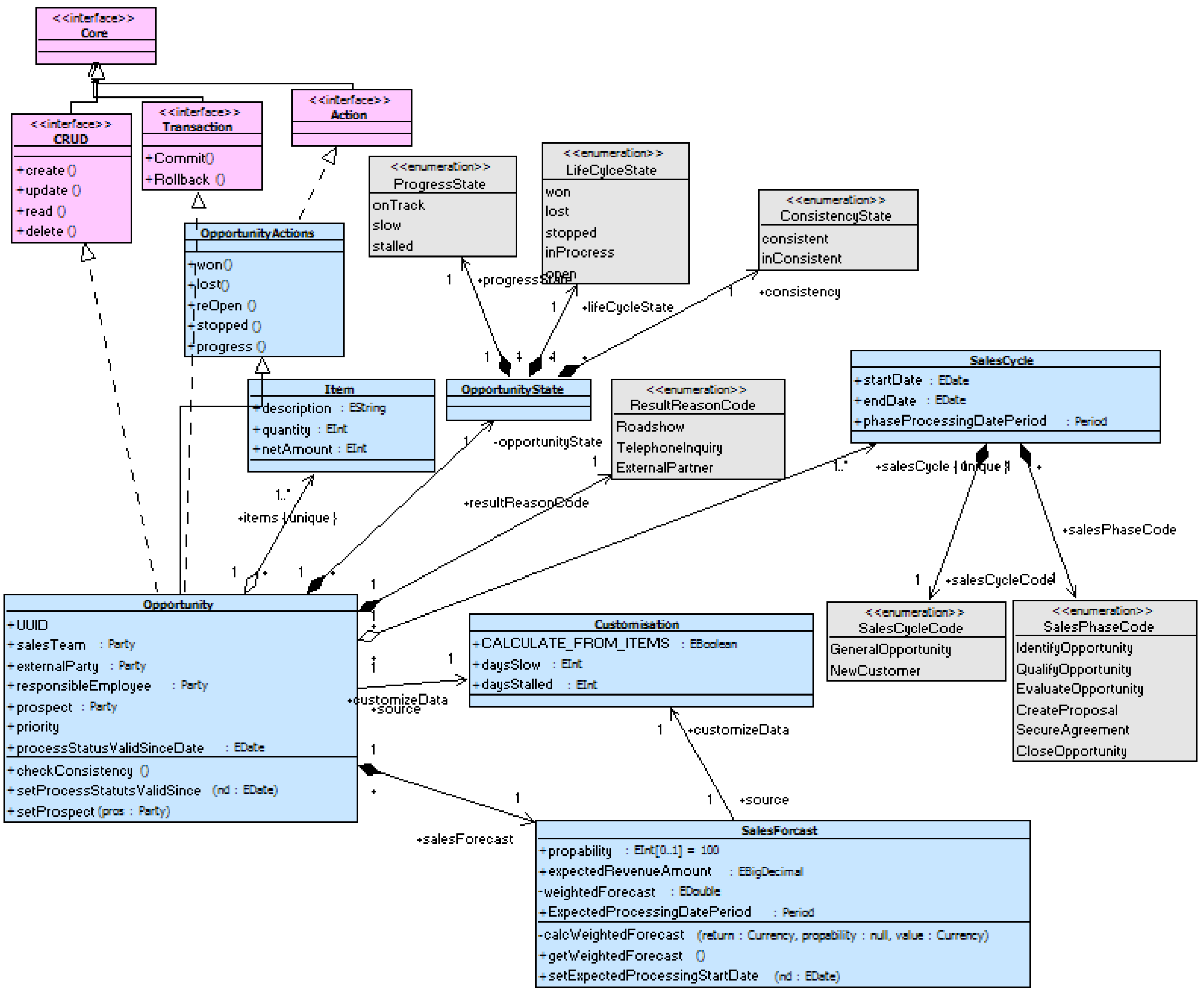

Figure 4: Main Classes in Opportunity Management

3.2 Crosscutting Concerns in the CRM application

In the following, we introduce *consistency checks* and *partner determination* as two examples of crosscutting concerns in opportunity management. We will elaborate in more details on the consistency checks example.

3.2.1 Consistency checks

Several consistency checks have to be performed when the state of the opportunity object or some of the associated objects changes. These consistency checks are crosscutting because they cut across different classes, i.e., the same checks need to be performed when attributes of objects that are defined in different classes change. Consequently, the code that enforces them is scattered across the implementation of several classes.

We classify the consistency rules into two types according to the degree of crosscutting.

a) Simple constraints

Constraints that involve only one business object class are called simple constraints. For instance, the simple constraints C0, C1, and C2 define an opportunity as being inconsistent if e.g., one of the following conditions is true:

- (C0) Opportunity.processStatusValidSinceDate > CURRENT_DATE
- (C1) SalesForecast.expectedProcessingDatePeriod.EndDate is not set
- (C2) SalesForecast.expectedProcessingDatePeriod.EndDate < SalesForecast.expectedProcessingDatePeriod.StartDate

b) Complex constraints

Some consistency constraints are called complex because they involve more than one business object. The enforcement of such constraints in an object-oriented design will be scattered across at least two classes. Constraints C3 and C4 are complex constraints, which specify when an opportunity is inconsistent:

- (C3) Opportunity.processStatusValidSinceDate < SalesForecast.expectedProcessingDatePeriod.StartDate

- (C4) SalesCycle.phaseProcessingDatePeriod.StartDate < SalesForecast.expectedProcessingDatePeriod.StartDate

Appropriate logic is needed to check such consistency constraints and hinder their violation. This logic should be triggered when the fields corresponding to the constraints are modified and also when the setter methods of these fields are called. For instance, to enforce the complex constraint C3, appropriate logic is required in the method *setProcessStatusValidSinceDate* of the class *Opportunity* to check that the date is smaller than *expectedProcessingDatePeriod.StartDate* in the associated *SalesForecast* object. Moreover, similar logic is needed in the method *setExpectedProcessingDatePeriod* to verify that the *StartDate* of the new period is smaller than the value of the attribute *processStatusValidSince* in the associated *Opportunity* object. Below, we show an implementation of these two methods in Java.

```
//defined in class SalesForecast
public void setExpectedProcessingStartDate (Date nd)
{
    if(nd > this.opportunity.processStatusValideSinceDate)
    this.expectedProcessingStartDatePeriod.startDate = nd;
}


//defined in class Opportunity
public void setProcessStatusValidSince(Date nd)
{
        if(this.salesForecast.expectedProcessingDatePeriod.startDate > nd)
        this.processStateValidSinceDate = nd;
  }
```

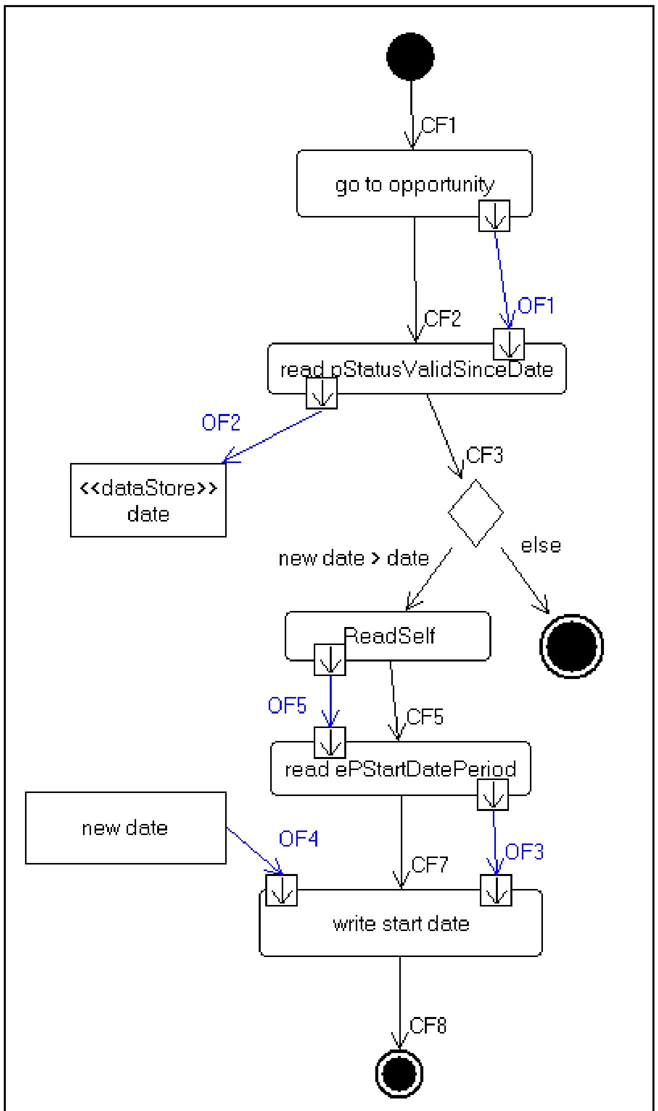

Figure 5: Method setExpectedProcessingStartDate

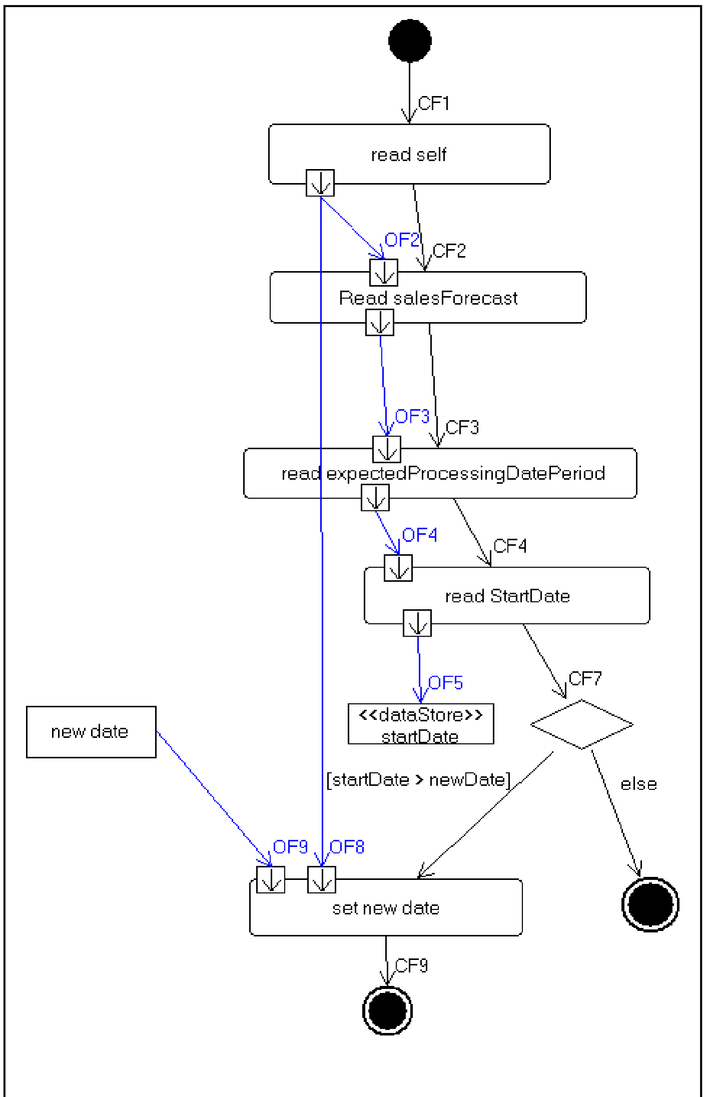

Figure 6: Method setProcessStatusValidSince

Figures 5 and 6 show the behaviour models that correspond to the method bodies of these two methods using UML actions. These models were drawn using the tool TopCased [8]. Figure 5 shows the body of the method *setExpectedProcessingStartDate*, which is defined in the class *SalesForecast* whereas Figure 6 shows the body of the method *setProcessStatusValidSince*, which is defined in the class *Opportunity*.

We observe that these models are complex. Moreover, the parts of the behaviour model that are responsible for setting the date attributes are mixed with the parts that implement the consistency check. Consequently, the consistency check cannot be modified without understanding the whole behaviour model. In addition, if the consistency check should be modified for some reason, then the user would have to find out the different model elements that are related to that check and modify all of them consistently.

Figures 7 and 8 show the behaviour models for the same methods as in Figures 5 and 6. However, these models do not contain any logic for checking/enforcing consistency rules, i.e., the method body just sets the appropriate attribute to the new date that is passed as parameter. One sees easily that these behavioural models are much simpler than those in Figures 5 and 6.

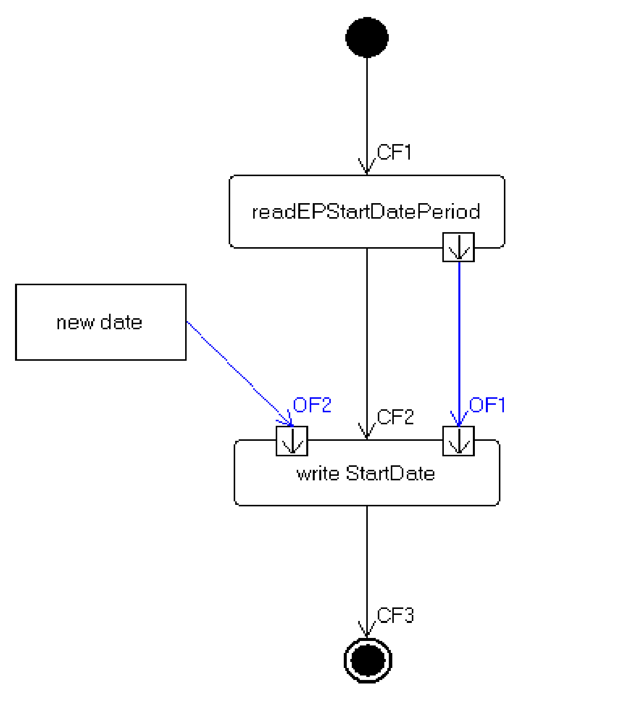

Figure 7: Method setExpectedProcessingStartDate

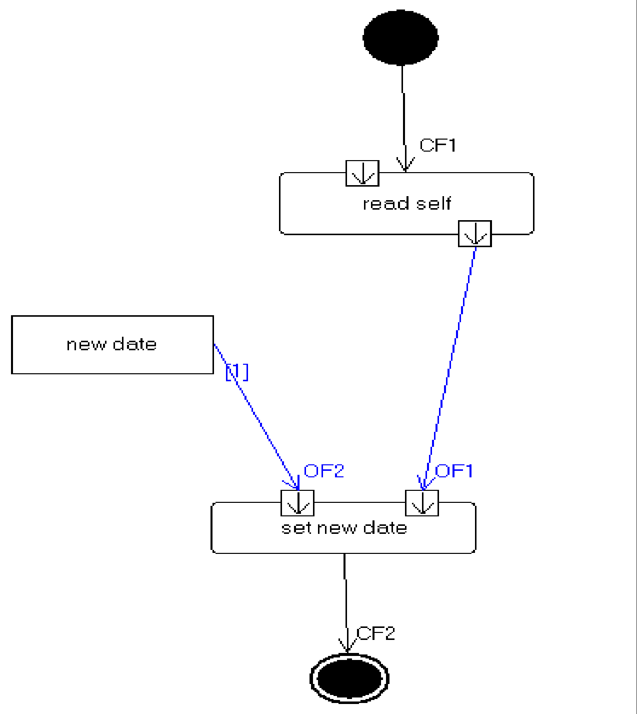

Figure 8: Method setProcessStatusValidSince

The java code corresponding to the models shown in Figures 7 and 8 is shown below.

```
//defined in class SalesForecast
public void setExpectedProcessingStartDate (Date nd)
{
    this.expectedProcessingStartDatePeriod.startDate = nd;
}
//defined in class Opportunity
public void setProcessStatusValidSince(Date nd)
{
    this.processStateValidSinceDate = nd;
}
```

3.2.2 Partner determination

Partner processing is a function in many SAP business applications (including CRM applications) that allows users to define partners with their company's terminology and specify how the system works with those partners. Partner processing ensures the accuracy and consistency of partner data, e.g., it can be used to make sure that an order document contains a ship-to-party. Without this field the order would be incomplete and the system cannot process it. Moreover, partner processing makes users work more easily with the software through advanced features such as partner determination.

Partner determination [9] refers to the system ability to automatically find and enter partner information such as addresses in certain transactions and documents. That is, the user enters manually one or more partners and the system determines and completes other partners and information by using several sources of information such as the business partner master data, the company organizational data, documents related to the current document, etc. Automatic partner determination can also be used to hinder users from entering inconsistent information or information that is already known to the system.

Figure 9 shows an example that illustrates how partner determination works. The user creates an opportunity and enters the name of the sales prospect and the system enters the name of the contact person (by checking the partner master data), the address of the sales prospect, and the name of the responsible employee for this opportunity (using the company organizational data).

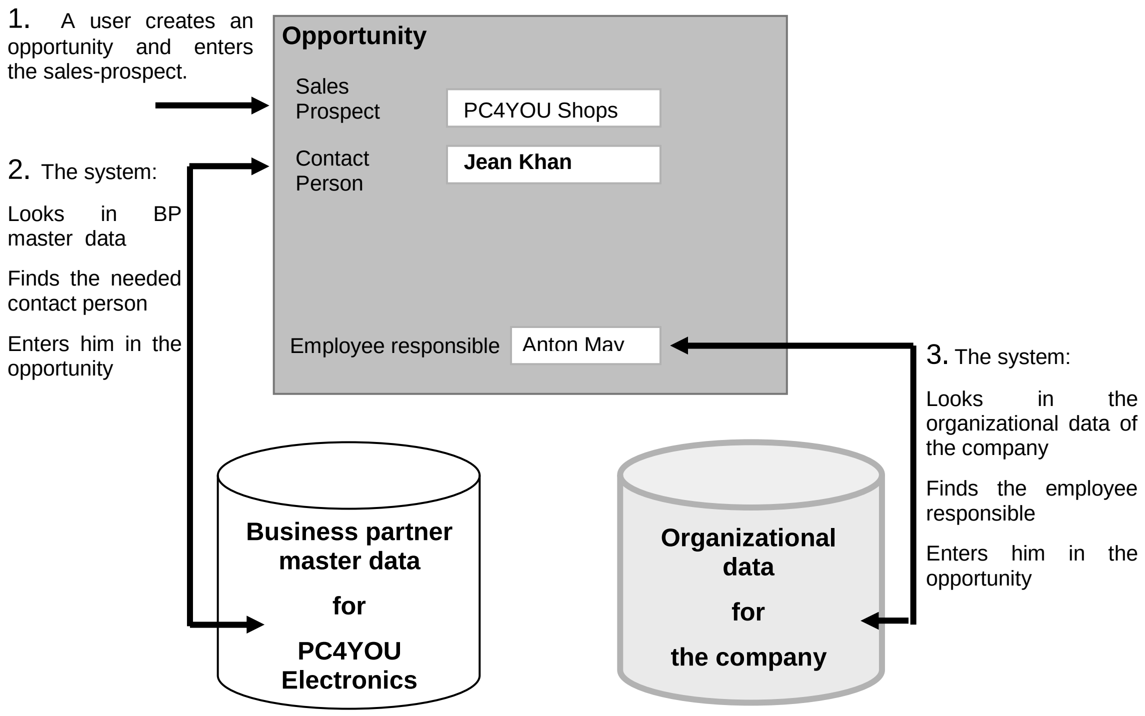
1. A user creates an opportunity and enters the sales-prospect.
Opportunity
Sales Prospect
PC4YOU Shops
Contact Person
Jean Khan
2. The system:
Looks in BP master data
Finds the needed contact person
Enters him in the opportunity
Employee responsible
Anton May
3. The system:
Looks in the organizational data of the company
Finds the employee responsible
Enters him in the opportunity
Business partner master data for PC4YOU Electronics
Organizational data for the company

Figure 9: Partner Determination in Opportunity Management

The way partner determination is done can be very different depending on the business process, the business transaction, and the companies that run the CRM software. Customers that use SAP CRM solutions can setup rules for the system defining what data sources to use for each *partner function* (e.g., contact person, sold-to-party, ship-to-party, etc.) and in what order these sources are searched according to their needs. They can also configure when partner determination is performed, e.g., when data is entered by the user or when the data is saved. These rules are called *partner determination procedures* [10] and they can be associated with a certain type of transactions (e.g., creating a new opportunity).

Partner determination procedures bring together *partner functions* and *access sequences*. That is, for each business function the user specifies whether automatic partner determination is needed and if that is the case he specifies a search strategy that defines where to search for partners for that partner function and in what order. This search strategy is refereed to as *access sequence*. For example, an access sequence can indicate the search order: preceding document, then the organizational data model, and finally some customer-specific function.

Partner determination is also a crosscutting concern. In fact, the code that is responsible for it is scattered across several classes of the user interface and the business objects of the CRM application. Automatic partner determination may be triggered in the UI classes e.g., when the partner enters the sales prospect for a new opportunity and can be also triggered in the business object opportunity when that object is saved. Partner determination functionality is scattered over other classes in the CRM application such as *SalesQuote* and *SalesOrder*.

In the following, we concentrate on the partner determination functionality in the business object opportunity. Partner determination is triggered when the sales prospect is entered (for the sake of the simplification we assume that this is done when the method *setProspect* is called) and also when the opportunity is saved (i.e., the method *update* of the CRUD interface is called). Below we show the implementation of these methods in pseudo java.

```
//defined in class Opportunity
public void setProspect (Party pros)
{
    this.prospect = pros;
    //run partner determination procedure for business function sales prospect

}
//defined in class Opportunity
public void update()
{
    //run partner determination procedures for update opportunity transaction
    //save the updated opportunity
}
```

Partner determination is a function that is characterized by several extensibility requirements. That is, customers should be able to define partner determination procedures and new access sequences according to their needs. To make such extensions easy and non-invasive, it is important to have the partner determination functionality well-modularized and separated from the other application logic.

3.3 Benefits of VIDE Aspects in modelling Business Applications

VIDE introduces Aspect-Oriented Software Development concepts to Model-Driven Software Development by defining new modelling constructs at the PIM level such as aspect, pointcut, and advice. Supporting aspects at the modelling level brings several benefits to the users that model business applications as explained in the following.

- ***Better modularization of crosscutting concerns at the model level*:**

VIDE allows defining fully executable applications through the use of UML actions. Consequently, VIDE models are more complex than traditional object-oriented models, which do not model any behaviour (i.e., the bodies of methods and constructors). This high complexity can be seen in the models presented in Figures 4 and 5.

To master the complexity of business application models, good modularization techniques are needed and aspect-oriented modelling is such a technique. Aspects provide means to modularize the logic belonging to crosscutting concerns. Thus, this logic will be defined in a separate aspect module rather than being scattered across several tangled models. Consequently, the complexity of business application models is reduced.

- ***Easier understanding and maintenance of models:***

Through the better modularization of crosscutting concerns, business application models become simpler and consequently easier to understand and to maintain. For instance, to modify a certain consistency check without aspects, the user would have to first identify all models and model elements (i.e., various classes, activities, etc) that are related to that consistency check. Then, the user has to update the models (e.g., by adding some fields and methods, or modifying the implementation of some methods) to accommodate the change of the consistency check. As several consistency checks are crosscutting, the same change needs to be done at multiple locations, which is quite redundant and error-prone. With aspects, this becomes easier because the user needs only to modify the consistency check aspects.

- ***Improved reuse of business logic:***

With aspects, crosscutting business logic can be encapsulated in separate modules and it is in this way no longer scattered across various model elements. Consequently, that business logic can be reused more easily.

Quantification is an important property of aspect-oriented approaches [11]. It refers to the ability to quantify over a set of points in the execution of a program in the case of Aspect-Oriented Programming (respectively a set of model elements in VIDE PIM models). This property is supported through the pointcut construct, which can be easily extended to select more join points (in our case model elements) so that a piece of crosscutting business logic can be activated and executed at multiple locations. For example, if some consistency check is modularized in an aspect and one wants to reuse that check in the UI classes in addition to the backend business object classes, then one only has to modify the pointcut of the consistency check aspect.

- ***Easier extensibility:***

Aspect-oriented software development provides techniques and constructs that support an easy extension of business applications. For instance, customers can define new partner determination procedures and access sequences in a modular and non-invasive way when partner determination is modelled as an aspect.

The pricing module can also be easily extended with customer-specific policies and rules by using aspects. This pricing module is used in sales quotations to determine the price whilst taking into account all discounts that the customer qualifies for. When pricing policies are modularized using aspects customers will be able to activate/deactivate them in a flexible way according to their needs. Moreover, new policies can be easily supported by modelling an appropriate aspect.

- ***Easier customization through better modularization of features:***

Aspects can be used as encapsulation modules for certain features. Through the composition mechanism, feature aspects can be easily switched on/off and composed with the application. Features are especially relevant in the context of product line engineering, where variants of a business application (e.g., a set of CRM applications) share several commonalities.

Product line development requires support for feature-oriented development at all stages, whereby a feature is an increment in program functionality. Features are a de-facto standard in distinguishing the individual programs in a product line, since each program is defined by a unique combination of features. Product-line architects reason about programs in terms of their features.

SAP is involved in several research projects, which aim at using AOSD and MDD in the context of Product Line Engineering such as the EU project AMPLE [13] and the national research project FeasiPLE [14]. The aim of AMPLE is to provide a software product line development methodology that offers improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their (forward and backward) traceability during product line evolution.

- ***Support for multiple target platforms and programming languages***

VIDE supports aspects at the PIM level, i.e., VIDE aspect-oriented concepts are not dependent on a specific aspect-oriented language such as AspectJ or AspectC++. Thus, VIDE aspects can be modelled once and mapped to multiple aspect-oriented programming languages with appropriate model transformations.

This particular benefit of aspect-oriented modelling in VIDE is especially important for SAP business applications because two application stacks (ABAP and Java) coexist together in some SAP products. With VIDE aspects, crosscutting concerns such as consistency checks and partner determination can be modelled once and then generated using appropriate transformations either as Java aspects (for consistency checks in the User Interface) or as ABAP enhancements (for consistency checks in the backend).

- ***Getting the benefits of AOSD without using an AOP language***

The aspect composition mechanism of VIDE can be implemented in various ways. In vertical composition, the object-oriented models are transformed to object-oriented code (e.g., in Java) and the VIDE aspects will be mapped to aspects (e.g., in AspectJ). In horizontal composition, which is the chosen composition approach in WP3, the composition of aspects and base application is done at the PIM level and the resulting model is purely object-oriented. Consequently, the resulting model can be transformed to object-oriented code. The horizontal composition alternative brings the benefits of AOSD to the model-level without posing any restrictions on the target programming language/platform.

3.4 Summary

In this section, we introduced a CRM application from SAP as an example business application. Then, we focused on opportunity management and discussed two crosscutting concerns there: consistency checks and partner determination. When modelling such concerns with aspects, several problems arise. These problems can be solved by using the aspect-oriented modelling capabilities of VIDE.

4 Realizing Aspect Oriented Composition in VIDE

This chapter describes a proposal for the integration of aspect-oriented concepts into MDD in the VIDE context. The realisation is a proof-of-concept solution and does not serve as a prototype or a separate tool to be used by users for modelling and composing aspect-oriented constructs. The main objective of the realisation, called *Demonstrator*, is to show the feasibility of the proposed concepts and to indicate possible technological problems.

Generally the realisation can be split into two parts. The first part contains the UML Profile extension for modelling aspect-oriented constructs. The second part provides a set of model-to-model transformations for implementing aspect composition at the PIM level.

After a short overview, the following sections present the required metamodel extensions and the transformations by means of some examples.

4.1 Overview

To allow having multiple model-to-code compilers in VIDE, the aspect composition should be implemented as a pre-processing step prior to the code generation by the model-to-code compiler For this purpose the concept of the horizontal composition has been chosen (see Figure 10). The models before and after the aspect oriented composition are models on the PIM level. Therefore the resulting model can be processed like the original base model without requiring any special handling of the aspect composition that was done before.

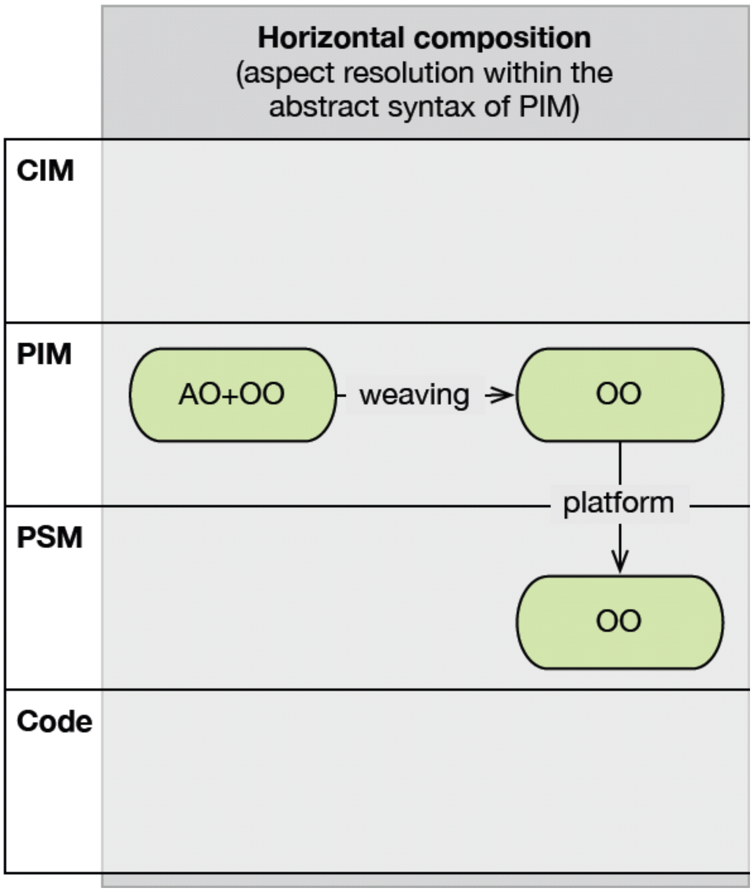

Figure 10: Horizontal Composition

In the first phase of the Demonstrator development, we extend the pure UML metamodel for integrating the aspect-oriented model. The VIDE/UML metamodel can be extended in the same way later on.

The concept provided by the UML Actions Semantics allows the modelling of behaviour using pre-defined model elements like actions and activities. The usage of the UML Action Semantics makes the control and data flow explicit. There is also the possibility, partially used in the VIDE context, to model behaviour using OCL expressions.

As already mentioned, the Demonstrator mainly consists of UML Profile extensions and a set of transformations realising the composition of the aspect and the base model. The transformations used for the aspect model composition require access to the modelled control flow to determine the interaction points and insert or modify the modelled behaviour.

The development of the required transformations based on UML Action Semantics proves to be the easier way because of the modelled behaviour is more explicit. Therefore the Demonstrator supports aspect composition works on behaviour models that are based on the UML Action Semantics, so we can mainly focus on the exploration of different composition strategies.

The following steps are necessary to extend UML (and later UML/VIDE) by aspects:

- The UML model is extended for modelling aspects.
- Model-to-model transformations for converting the aspect model to a plain VIDE/UML model are developed.
- The resulting model can be fed into any VIDE-to-code generator, the aspect behaviour is executed by generated VIDE/UML model elements.

4.2 Technological overview

Regarding the technologies to be used for an integrated aspect-oriented and model-driven scenario, there are mainly two influencing forces: a) the given technology from the underlying model-driven infrastructure and b) the additional technology needed to support the aspect-oriented extensions. In the context of the VIDE project, the underlying dependent technology is the model repository with its metamodels and the editors used to create user models. Following a horizontal approach where aspect composition is realized completely at the PIM level no direct integration with the model compiler is necessary. This is, because the aspect composition is implemented solely as an external tool component working on the model repository. As such, it adds one additional step to the “build process” – right before a model compiler generates code from the (composed) models. A vertical approach on the other hand would require the aspect composition to be developed as part of the model compiler. Support for aspect-oriented modelling also has to be incorporated into metamodels, notations and tools like editors. Further explanation of architectural issues will follow as part of WP8.

4.2.1 Base Technology

UML 2.1

The metamodel of the VIDE language is based on UML 2, supporting a subset of its structural and behavioural modelling capabilities. On top of that metamodel, there will be at least one textual and one graphical concrete syntax, simplifying the construction of VIDE models.

Extensions to the original UML 2 metamodel are realized as UML Profiles and are described in Deliverable 2.1.

In the absence of usable VIDE model editors during the development of the WP3 demonstrator, other modelling tools had to be used for the example models. For this purpose, IBM Rational Software Architect and Topcased were chosen, because of their UML 2 Activity modelling facilities.

OCL 2.0

As the final VIDE language will make use of OCL to directly support queries and to ease the specification of navigation and read access to features in Activities, an integration with the aspect models and composition process has to be carried out. The current approach and demonstrator do not directly support OCL in VIDE models yet. While the replacement of read Actions with corresponding OCL statements should be straight forward from a technical point of view, the integration with queries would require more research, because they may introduce new join point kinds and composition issues.

4.2.2 Aspect Extensions

Supporting aspect-oriented concepts in a model-driven environment requires the following components to be integrated with existing technology:

Aspect Modelling

The modelling infrastructure has to be extended to support the expression of aspect-oriented language constructs in user models. These extensions should integrate seamlessly with existing model elements to be used in an intuitive and productive way. Technically, the introduction of aspect-specific modelling constructs involves changes in the abstract syntax (i.e. the metamodel) and the concrete syntax (i.e. the notation that is mostly hard-coded in model editors). These changes should be completely additive, leaving the base modelling language independent from the aspect-oriented extensions.

Pointcut matching

The next essential step in an aspect-oriented composition process is the evaluation of pointcuts, and thus identifying join point shadows in the base models. The functionality of the pointcut matching itself has to be incorporated on the tooling side and requires parsing pointcut expressions and querying the base models. In a 2-step composition process, as proposed in our demonstrator, an additional meta-model extension is needed for the annotation of located join point shadows in an intermediate model.

Another - optional - feature could be the visualization of identified join point shadows and their crosscutting (or the impact of the aspect) in the base model. Such visualization would have to be implemented in the notational tooling, that is, the model editors.

Aspect composition (weaving)

The final composition step involves the weaving of aspect elements (structure, advices) at the join point shadows as well as the generation of and integration with aspect infrastructure. This step requires navigation and analysis of the base and aspect models and finally the modification of the base model according to the chosen weaving strategy and the aspect model.

4.2.3 Requirements

The implementation of these features imposes some technological requirements.

For metamodel extensions as needed for aspect models and join point shadow annotations, the underlying modelling language should support easy and purely additive metamodel extensions. This can be achieved by using an elaborated metamodel infrastructure like MOF/EMF in conjunction with UML 2.1. The metamodel should support the annotation of model elements, including references to other model elements. For pointcut matching and aspect weaving, the navigation/query capabilities of models are important and should be supported by either the metamodel API or an external query language.

On the tooling side, pointcut matching and aspect weaving can be realized with existing model-to-model transformation technology. A suitable transformation technology should offer good navigation and querying capabilities including quantification over properties of model elements. It should be possible to add missing functionality through custom extensions, like e.g. user-defined query functions. Considering the nature of pointcut expressions, a declarative transformation language should fit better than operational/imperative ones. In the case that base and aspect models are kept separate, the model transformations have to be able to deal with multiple input models with potentially different metamodels. Using UML 2 Profiles for light-weight metamodel extensions, the transformations must be able to handle them in source and target models, i.e. analyzing and creating stereotype applications correctly. Modularization and combination/layering of model transformations could be an issue when they get complex and/or different weaving strategies are to be supported in the transformation process.

To integrate with the VIDE architecture, the AO composition module has to work with the underlying model repository and later be integrated in user tools (mainly editors). As the VIDE model repository is based on Eclipse UML2, the EMF-based implementation of UML 2, the aspect composition module should also be build on that base.

4.2.4 Model Transformations

In MDA, model transformations are used for two different purposes. The first and most used kind are transformations that map platform independent models (PIM) onto a platform, creating a platform specific model (PSM). The property “platform independent” is relative, i.e. it is a technical refinement step that can be done incrementally, each time adding more platform specific detail to the application model. Ultimately, a last transformation typically produces source code artefacts for the target platform, thus leaving the modelling world.

The second purpose of model transformations is not so widely used, but in the context of aspect-oriented modelling of great importance. These transformations do not alter the level of platform (in)dependence, but refine the model according to other concerns. These concerns come either from the domain or the application itself.

Model transformations that produce target models from other models are called *model-to-model* (M2M) transformations, whereas transformations producing textual artefacts (typically source code) are called *model-to-code* (M2C) transformations. The latter are often based on template languages, because of their syntactical affinity with the output artefacts. Model-to-model transformations on the other hand are usually realized on M3 level, allowing the transformation of models from arbitrary metamodels.

For the demonstrator of WP3, ATL (ATLAS Transformation Language, [42]) was chosen as model transformation language, because of its maturity, support for multiple input and output models and metamodels, OCL navigation and querying syntax and ability to process UML2 models. ATL is a hybrid transformation language, supporting declarative as well as imperative transformation rules. This offers enough flexibility to implement even complex pointcut matching and aspect composition rules.

4.3 AOC Architecture

The architecture for the aspect-oriented composition at PIM level describes UML Profile extensions as well as transformation processes (see Figure 11).

The composition process “merges” the input models embarking on a strategy. The input of the AOC process is a base model and an aspect model. The base model is a plain UML/VIDE model. Located in the aspect model are the aspects which extend the behaviour and/or the structure of the base model elements.

The transformation process is shown in Figure 11. It is split up in two phases: *Pointcut Resolving* and *Aspect Weaving*. This is done to allow changing the weaving phase independently of the pointcut matching phase (e.g. to implement a different strategy) and for debugging purposes. These two phases are quite independent. The final output of the AOC transformation process is a plain VIDE/UML model.

As depicted in Figure 11 the pointcut matching transformation produces an intermediate model, which is the input base model enhanced with markers for the matched join points. Therefore not only a UML Profile for modelling aspect-oriented constructs, but also an UML Profile for marking model elements as join points is necessary. The intermediate model serves as the input model for the aspect weaving transformation, which inserts or modifies behaviour at the marked join points.

The next section provides a description of the input models and the corresponding required UML Profiles.

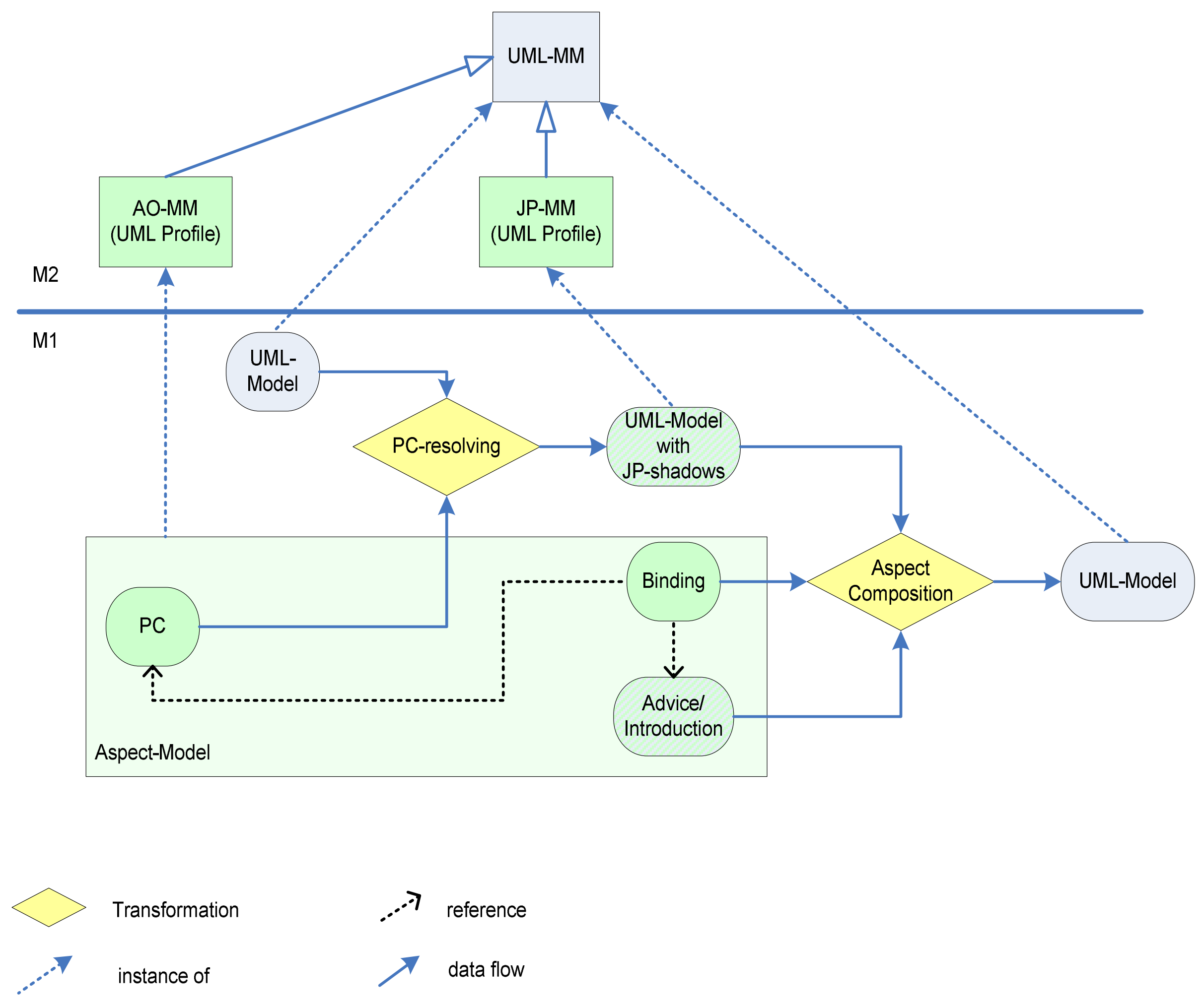

Figure 11: AOC Architecture

4.4 Models and Metamodels

4.4.1 Base Models

The base models are pure VIDE/UML models. To create base models, no further UML Profile is necessary.

Until further details are available from the other work packages about the exact structure and contents of VIDE PIM models, we assume the following:

- A PIM model consists of a structural part defining classes with their attributes and operations.
- The exact and complete behaviour of each operation is defined in one Activity which makes use of the model elements selected in the VIDE metamodel.
- Currently, the use of OCL Expressions instead of read actions is not supported.

4.4.2 Aspect models

Aspects are defined in separate models. They include pointcuts, advice and the association between a pointcut and an advice. The metamodel for aspect models is defined by a UML profile described in the next section. Advices encapsulate the additional behaviour and the pointcuts describes in a declarative way, where to insert or adapt the behaviour.

4.4.3 AO UML Profile

The AO UML Profile defines the modelling of aspect-oriented constructs in UML (see Figure 12). The following aspect oriented constructs are supported by this UML Profile:

- Aspect
- Advice
- Binding
- Pointcut

Aspect, Advice and Pointcut were already described in the section "Core Terms". The binding represents an association between an advice encapsulating the additional behaviour and the pointcut, which declares the locations (join points) for inserting that behaviour. Consequently an advice can be bound to more than one pointcut and a pointcut can be bound by more than one advice.

The AO UML Profile can be split into two parts: *Adaptation* and *Quantification*. There are elements defining *adaptation* (Figure 13) and *quantification* (Figure 14).

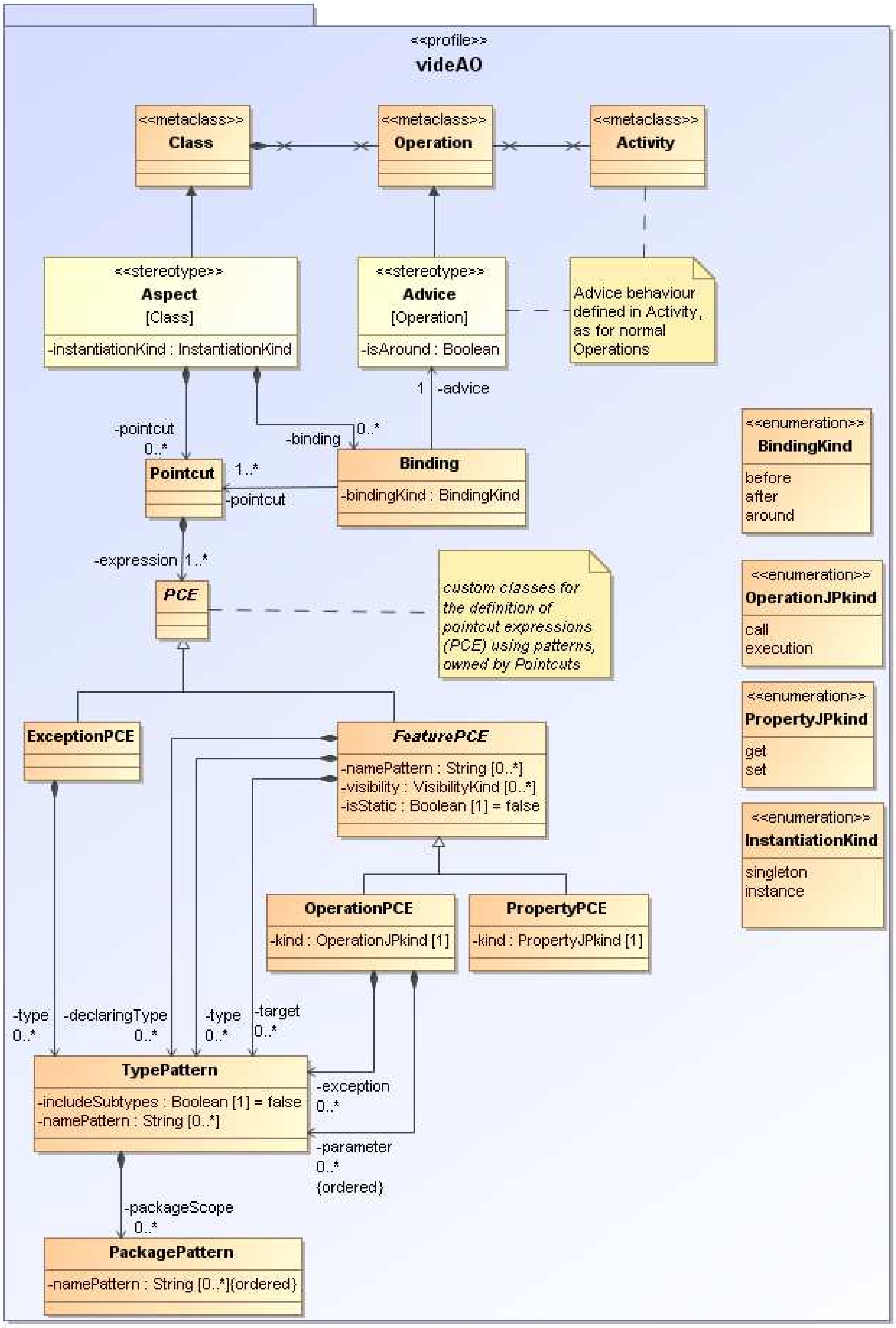
<<profile>>
videAO
<<metaclass>> Class
<<metaclass>> Operation
<<metaclass>> Activity
<<stereotype>> Aspect [Class]
-instantiationKind : InstantiationKind
<<stereotype>> Advice [Operation]
-isAround : Boolean
Advice behaviour defined in Activity, as for normal Operations
1 -advice
-pointcut 0..*
-binding 0..*
Pointcut
1..* -pointcut
Binding
-bindingKind : BindingKind
-expression 1..*
PCE
custom classes for the definition of pointcut expressions (PCE) using patterns, owned by Pointcuts
ExceptionPCE
FeaturePCE
-namePattern : String [0..*]
-visibility : VisibilityKind [0..*]
-isStatic : Boolean [1] = false
OperationPCE
-kind : OperationJPkind [1]
PropertyPCE
-kind : PropertyJPkind [1]
-type 0..*
-declaringType 0..*
-type 0..*
-target 0..*
TypePattern
-includeSubtypes : Boolean [1] = false
-namePattern : String [0..*]
-exception 0..*
-parameter 0..* {ordered}
-packageScope 0..*
PackagePattern
-namePattern : String [0..*]{ordered}
<<enumeration>> BindingKind
before
after
around
<<enumeration>> OperationJPkind
call
execution
<<enumeration>> PropertyJPkind
get
set
<<enumeration>> InstantiationKind
singleton
instance

Figure 12: AO UML Profile

4.4.3.1 Adaptation

The Aspect stereotype depicts a class as an aspect. The property *instantiationKind* describes how the aspect is to be instantiated. Singleton means there will be one aspect instance for the whole system, if *instantiationKind* is set to "instance", one aspect instance will be created for each object instance from which an advice is called. An advice stereotype classifies an operation as an advice operation which can be bound to pointcuts. Bindings state on which join points (selected by the pointcut property) an advice operation is executed. The property *bindingKind* of the Binding element states whether an advice operation will be executed. Supported binding kinds are before, after and around bindings.

Inside advice operations it has to be possible to access parameters, source and target objects of the join point. In an around advice a *ProceedAction* calls the original join point with its parameters and returns its return value.

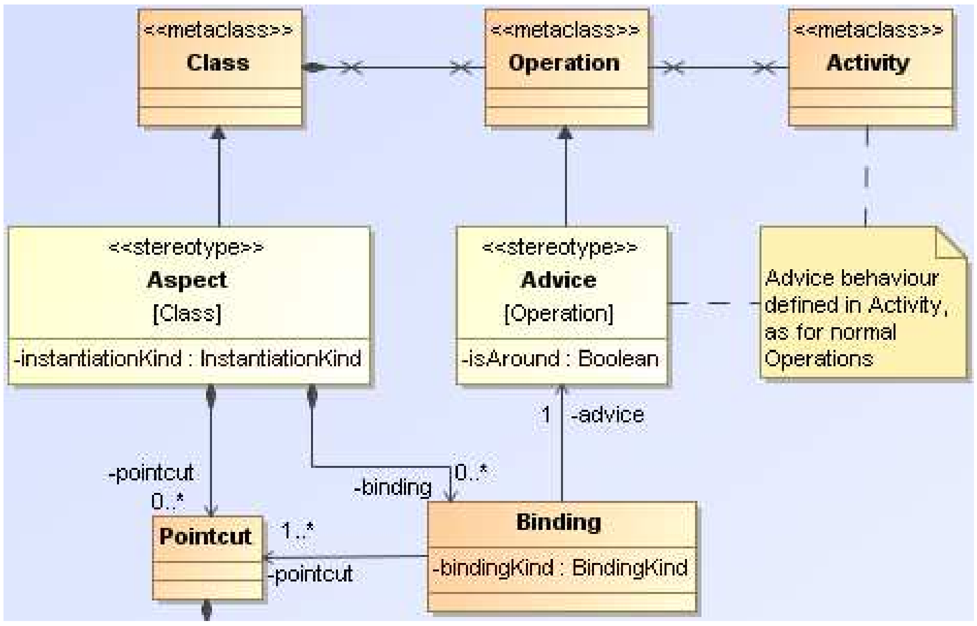

Figure 13: Adaptation part of AO UML Profile

4.4.3.2 Quantification

Model elements for quantifying join point sets are Pointcuts and PCEs (pointcut expressions). A pointcut has a *name* property and a property *expression*. The *expression* property holds the pointcut expression selecting the join points. The demonstrator supports operation expressions (*OperationPCE* element) for selecting method calls and executions as well as property expressions (*PropertyPCE* element) for selecting property read and write accesses. Both expression elements have the properties *namePattern*, *visibility*, *isStatic*, *declaringType*, *type* and *target*. A join point must fulfil all properties of a pointcut expression to be selected.

The *namePattern* property specifies the name of the selected feature, operation name and property name respectively. If multiple names are specified, a join point matches the expression if its name matches one of the name patterns. Name patterns can include the wildcards "?" and "*". The question mark is a placeholder for one arbitrary character; the asterisk matches any number of characters. Thus the string "Foo*ba?" matches "Foobar" and "Foofoobar", but not "Foofoofoo" or "Foobarr".

The *visibility* property restricts the selected join points to those with the set visibility (*private*, *protected*, *public*). Similarly the *isStatic* property omits static (if set to false) or non-static (if set to true) join points.

The properties *declaringType*, *type* and *target* specify different types in the context of join points. The property *declaringType* is the type owning a join point shadow (operation's or property's owner). Property *type* specifies the return type of the method or the property's type respectively. The property *target* specifies the type of the object on which an operation or property access is called. All three properties are optional and can specify multiple types. In the case when a property specifies multiple type patterns a join point must fulfil one of the type patterns to match the property (disjunction, "or" function).

The pointcut expressions are composed of some or all of these properties. Each specified property narrows the search scope for join point shadows. There is a conjunction ("and" function) between the specified properties of a pointcut expression.

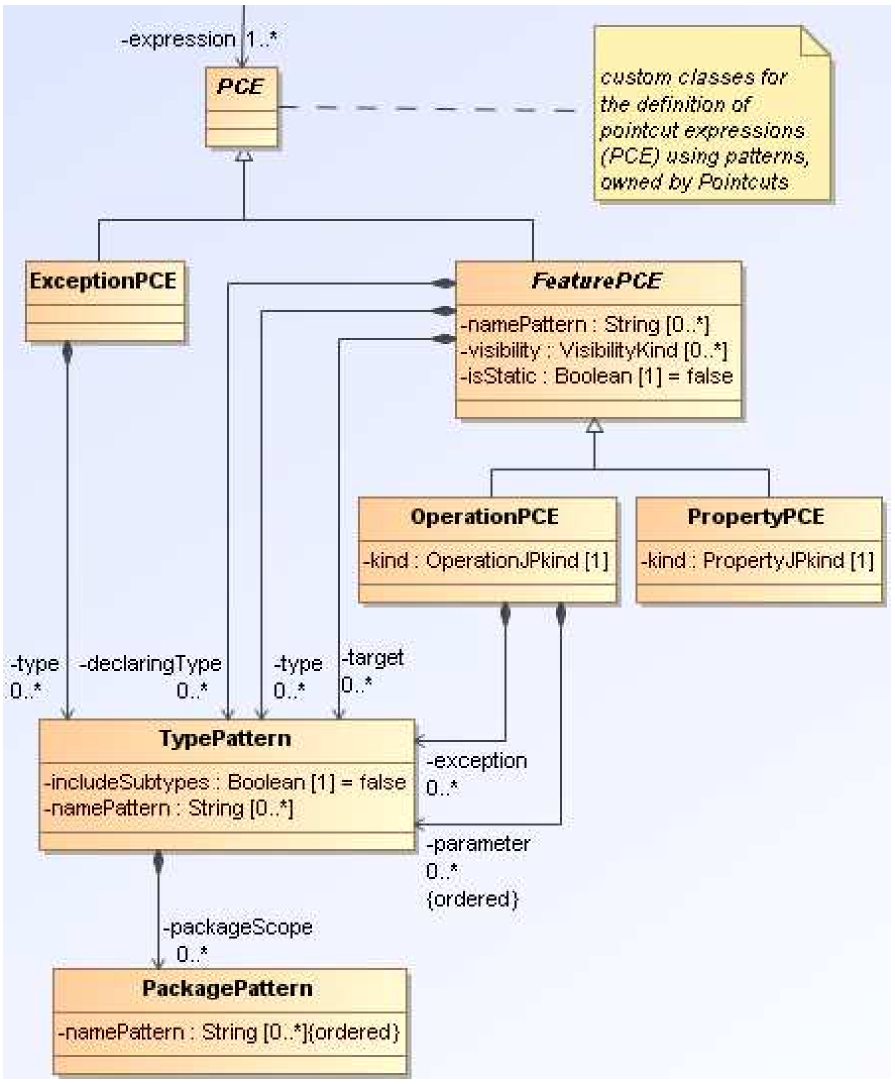
-expression 1..*
PCE
custom classes for the definition of pointcut expressions (PCE) using patterns, owned by Pointcuts
ExceptionPCE
FeaturePCE
-namePattern : String [0..*]
-visibility : VisibilityKind [0..*]
-isStatic : Boolean [1] = false
OperationPCE
-kind : OperationJPkind [1]
PropertyPCE
-kind : PropertyJPkind [1]
-type 0..*
-declaringType 0..*
-type 0..*
-target 0..*
TypePattern
-includeSubtypes : Boolean [1] = false
-namePattern : String [0..*]
-exception 0..*
-parameter 0..* {ordered}
-packageScope 0..*
PackagePattern
-namePattern : String [0..*]{ordered}

Figure 14: Quantification part of AO UML Profile

Currently supported join point kinds are *method call*, *method execution* and *field accesses* (get and set). These are probably the most common join point kinds. The quantification part of the AO UML Profile is easily extensible to implement new join point kinds or new aspect instantiation types. In a first step, the demonstrator only supports static pointcut expressions (no cflow etc.). Also, the definition of pointcuts is restricted to the use of standard tooling for UML Profiles, i.e. editing of tagged values (there is no special concrete syntax yet – neither textual nor visual). It should be found out if OCL can be used directly for the specification of advanced pointcuts (this is currently not easily possible in the demonstrator).

4.4.3.3 Example

To demonstrate the expressiveness of the modelled pointcuts, an example in AspectJ syntax is modelled using the AO UML Profile. The AspectJ syntax for this pointcut is as follows:

`set(* Date Period.*) || set(* Date Opportunity.*)`.

This pointcut expression is modelled in Figure 15.

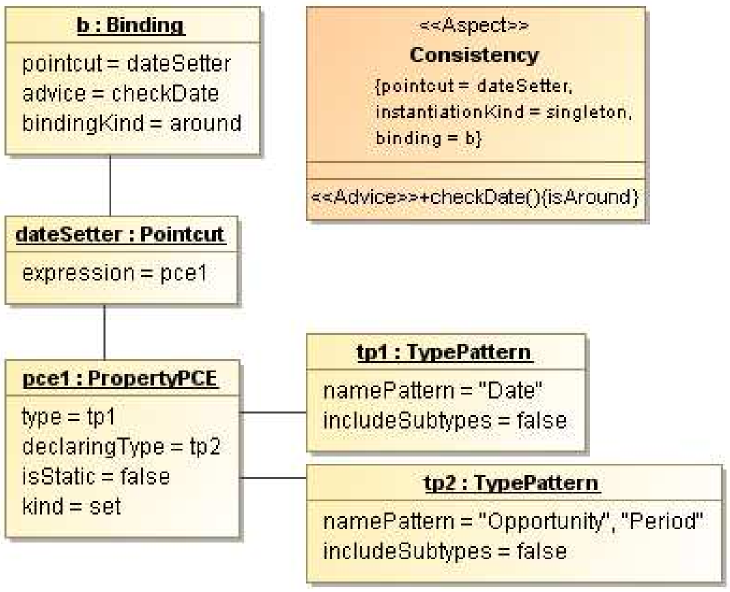

Figure 15: Example for modelling pointcuts using the AO UML Profile

4.4.4 Intermediate model: Join Point Shadow UML Profile

The result of the pointcut matching transformation is the join point shadow model. All base model elements matching pointcuts of the aspect model are annotated by a *<<JPshadow>>* stereotype. Each supported join point kind is represented by a separate stereotype to make the annotation explicit and to facilitate the handling of the intermediate model by the corresponding transformation. These stereotypes are defined in a separate UML Profile (see Figure 16).

The *binding* property of the stereotype holds the relation to one or more bindings in the aspect model. These bindings have properties pointing to the advice operation which is to be executed at the join points. The join point shadow model can be used for visualizations of the locations in the base model where aspects or advice operations get applied. It is also useful for debugging purposes when developing the model transformations.

In the final VIDE environment the join point metamodel is a mere intermediate model which is not visible to or editable by the user.

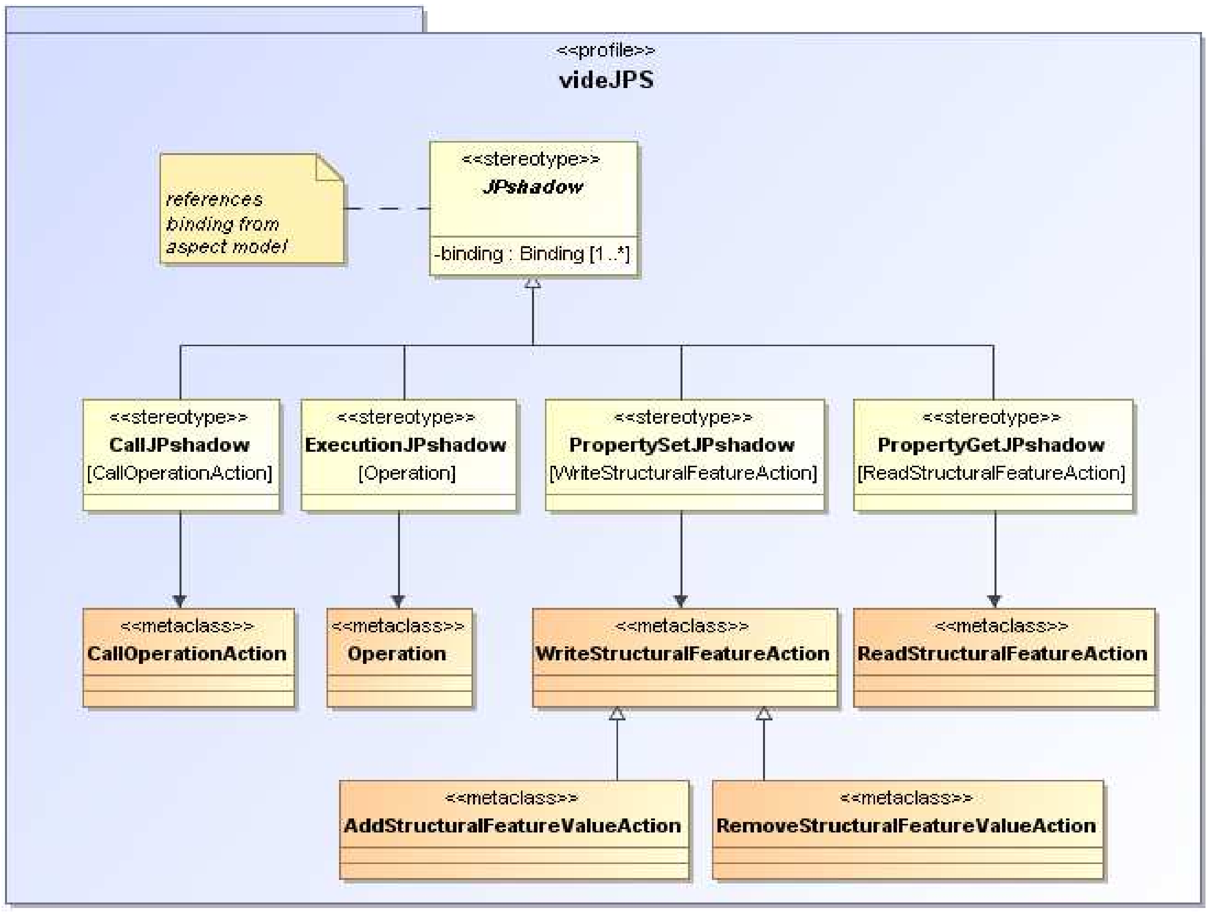

Figure 16: Join Point Shadow UML Profile

4.4.5 Result model

The resulting model is a pure VIDE/UML model at PIM level again. Usually the resulting model should not be changed or even seen by the user. This is because a lot of model elements are generated by the model transformations to implement the aspect behaviour.

These transformations, which are part of the aspect composition, are described in the following sections.

4.5 Model Transformations

After the extension for modelling aspect-oriented constructs was introduced, this section describes the model transformations which are required for realizing the aspect composition at the PIM level.

As already mentioned, the transformation process of the base and aspect models to a VIDE model is divided into two phases, pointcut matching and aspect composition (or aspect weaving).

4.5.1 Pointcut Resolving

The pointcut resolving transformation translates a base model (VIDE/UML model) and an aspect model into a join point shadow model (intermediate model). All pointcuts are resolved and their matching elements in the base model are annotated with corresponding stereotypes from the join point shadow UML Profile. To resolve pointcuts, elements from the base model are checked for properties declared in the pointcuts.

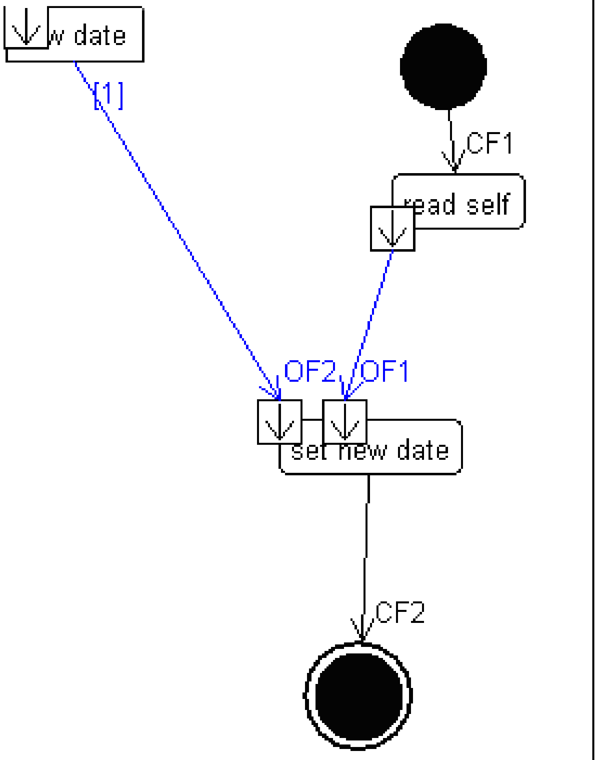

Figure 17: Model before Pointcut Resolving

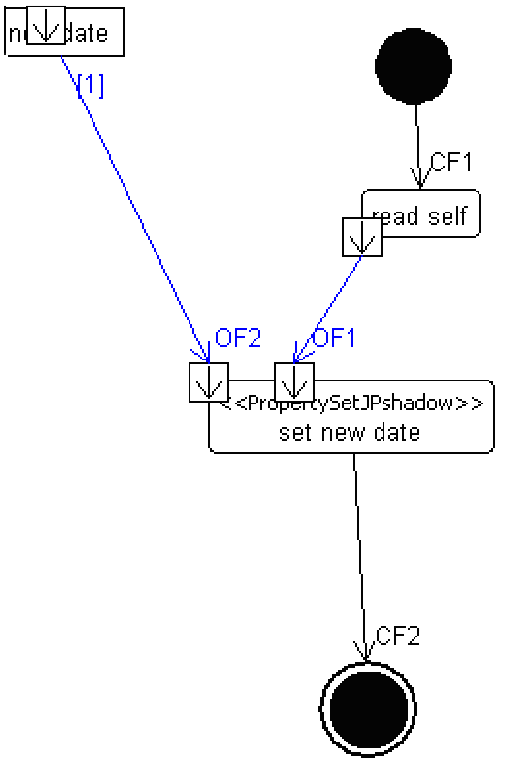

Figure 18: Model after Pointcut Resolving

The example pictured in Figure 17 and Figure 18 show models before (Figure 17) and after (Figure 18) pointcut resolving. While resolving the pointcut declared in Figure 15 the corresponding transformation generates the stereotype <<*PropertySetJPshadow*>> at the action node "set new date" in the model after pointcut resolving. After pointcut resolving has been processed the transformations responsible for aspect composition can access the information about the types of join point shadow.

4.5.2 Aspect composition

Aspect composition is also realized as a set of model-to-model transformations. These transformations weave aspect structures and advices into the annotated base model, resulting in a standard VIDE/UML model. This model is completely woven, i.e. all base and aspect behaviour is integrated and the model does not contain any aspect-specific elements. This resulting model can be processed by any transformation which expects the input of VIDE/UML models, e.g. model-to-code compilers.

The inputs of the aspect composition transformation are the join point model and the aspect model. This transformation replaces all join point shadows with model elements representing the aspect behaviour. Different transformation operations are needed depending on the join point kind, the binding kind and the aspect instantiation kind.

The chosen weaving strategy encapsulates the aspect in a separate class and creates calls to the advice operations on the matching join points. The advice operations are defined in aspect classes. By contrast in the approach of Fuentes and Sanches [21] the advice activity model is inlined at the join points. The approach generally allows an easier transformation in terms of special advice actions. For instance the *getTarget* action is transformed to the *readSelfAction*.

Because aspects are encapsulated in separate classes, there are possibilities within the Demonstrator to realize several instantiation strategies with a minimal effort. For different instantiation of the aspect class, generally we only have to change the creation mechanism of the current aspect class during the aspect composition. The creation of the aspect class can for example be realized using the *Singleton* pattern. In this case, the aspect class has only one instance and all calls to the encapsulated advice operations are called on the same instance. Different weaving strategies and their impacts, advantages and disadvantages will be discussed in *Deliverable 3.2*.

The transformation in our approach translates each element having a join point stereotype. If the join point is a UML action it generates the following model elements:

1. Action node(s) to retrieve the aspect instance (depending on aspect instantiation)
2. Action node(s) to invoke the advice operation (depending on advice kind)
3. Object flow edges to pass aspect instance to advice call
4. Edges to introduce actions into the control flow of the original action (depending on advice kind)

4.5.2.1 Advice invocation

If the advice is bound before or after the join points one *CallOperationAction* is generated. This *CallOperationAction* calls the advice operation given in the join point's binding.

If the advice is to be woven "around" (instead of) the join point the transformation must generate more model elements. In "around" advice operations it is possible to call the adapted operation with a *ProceedAction*. This join point can have parameters which have to be available at the *ProceedAction*. This is done by a closure object. The motivation for this approach was given by the Weaving Strategy in AspectJ (see [41]).

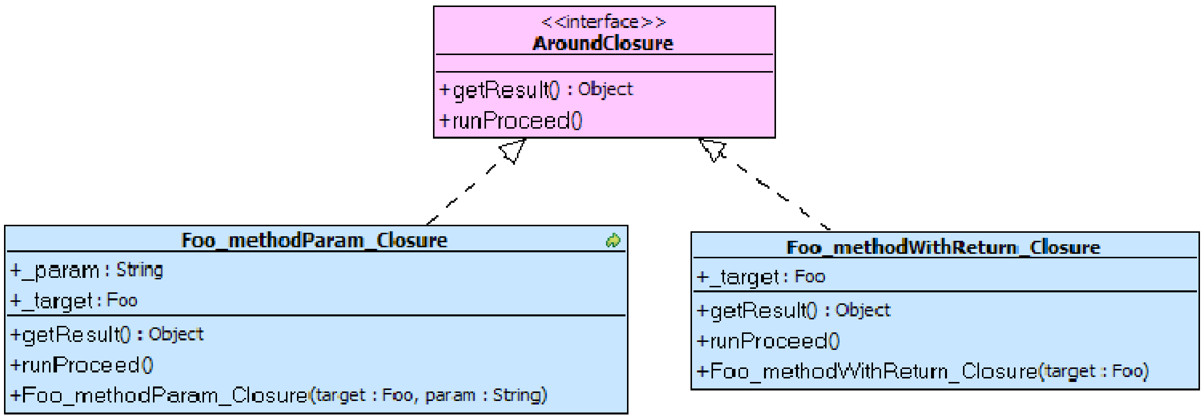

Figure 19: Examples for "Closure" classes

The closure stores the parameters of an operation call and its return value (see Figure 19). Therefore a closure class is created for each operation which can be a target of a call which is adapted by an “around” advice. The closure classes implement a common interface because one advice call can be called at different join points which can have different signatures.

On each join point a closure object is created which stores all parameters of this join point. This closure object is input of the generated advice operation call. Each *ProceedAction* within the advice operation is translated into a call of the closure’s *runProceed* operation.

4.5.2.2 Aspect Instantiation

For calling the advice operation the aspect instance must be provided. In AspectJ there are several aspect instantiation kinds, two of them are implemented in the Demonstrator: Singleton aspects (*issingleton* or no modifier), one aspect instance per current object (*perthis*). Singleton aspects are used when aspects monitor all objects of a kind or when aspects do not have to store information at all. *Perthis* aspects are used for object wrappers or when objects are to be augmented with additional data.

Possible future additions are the remaining AspectJ instantiation kinds like one aspect instance per called object (*pertarget* in AspectJ) or one instance per join point call.

4.5.2.3 Consistency check example: an around advice

In this section an example for an around advice is described. Starting with an object-oriented model, the model from Figure 6 (presented in section 3.2.1) an advice is extracted. The binding kind is *around*. Next the matching and weaving process is demonstrated. The first transformation annotates the matching join point shadows (matching). The second transformation translates the join point shadows in aspect behaviour in a plain UML model.

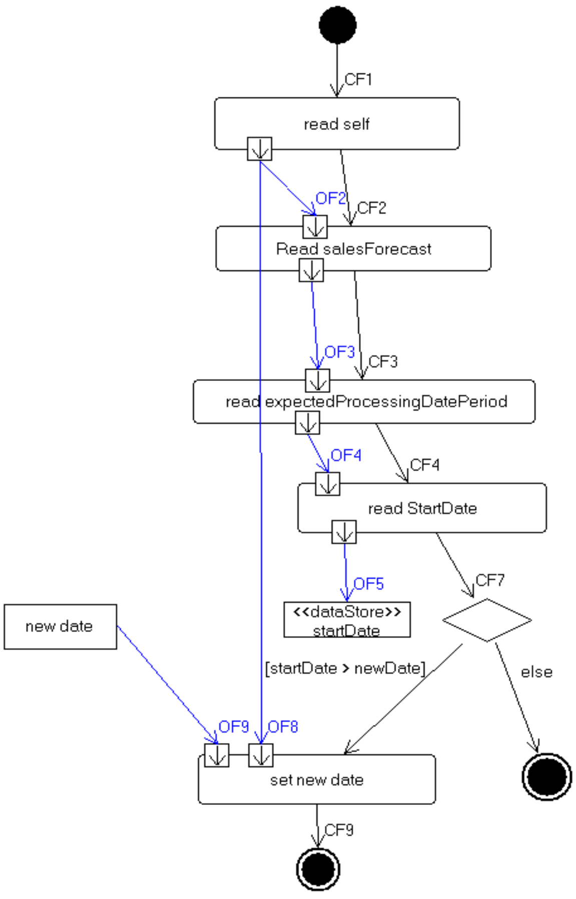

Figure 20: Method setProcessStatusValidSince

Figure 20 shows the model of an object-oriented method. This operation compares the property self.salesForecast.expectedProcessingDatePeriod.StartDate with the parameter newDate. If the former is bigger than the latter, the property self.processingStatusValidSince is set to newDate, otherwise nothing happens. This date check implements a consistency check. The purpose of the future advice is to extract all consistency checks into one aspect to have them in one module.

The extracted advice and the remaining base code are shown in Figure 22 and Figure 21 respectively. The base code now consists only of operations to set the new date while all other operations are enforcing consistency constraints and therefore must be moved into the advice. Inside the aspect the "proceed" action takes the place of the former "set new date" action. The "proceed" is a placeholder for the execution of the adapted join point shadow. In our case this is the "set new date" action. In that way, the logic for setting the date and enforcing the consistency check are separated.

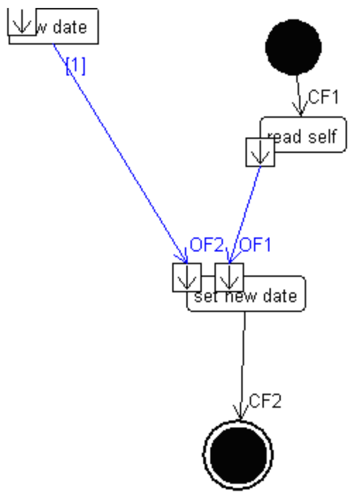

Figure 21: Base model

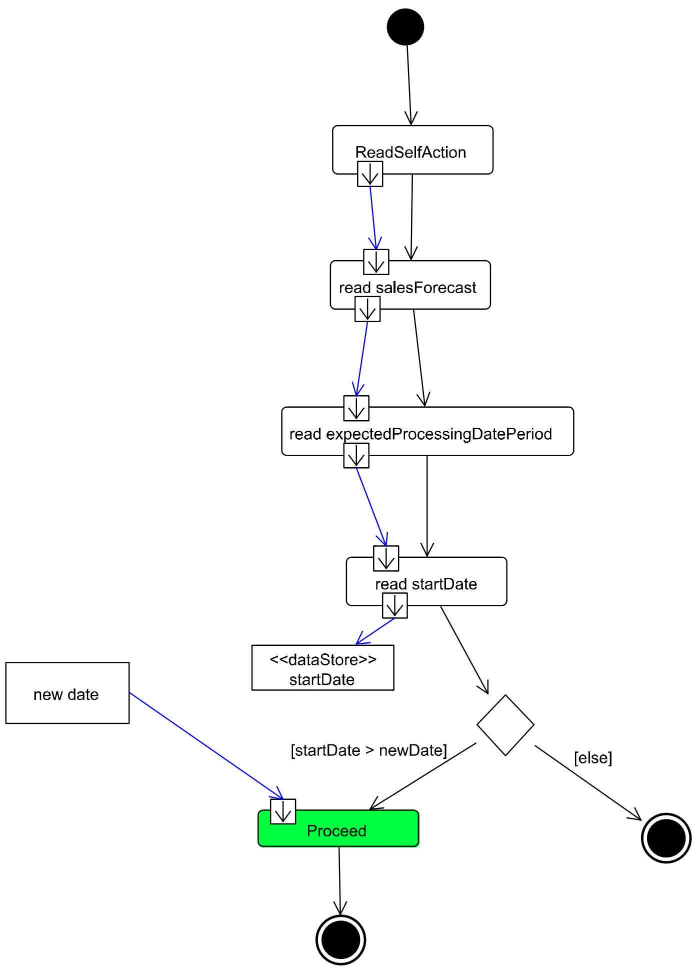

Figure 22: Advice in aspect model

The pointcut's purpose is to describe the join points in the base model on which a consistency check has to be performed. In the example this is the case when a date is set. The pointcut (shown in Figure 23) is similar to the one in section 4.4.2. It selects all write accesses (kind = set) of properties which have the type Date (pce1.type.namePattern = "Date") and which belong to objects of type "Opportunity" (pce1.declaringType.namePattern = "Opportunity").

The advice operation is bound “around” the selected join point shadows. This is because it replaces the original behaviour and executes it only when the date is “consistent”.

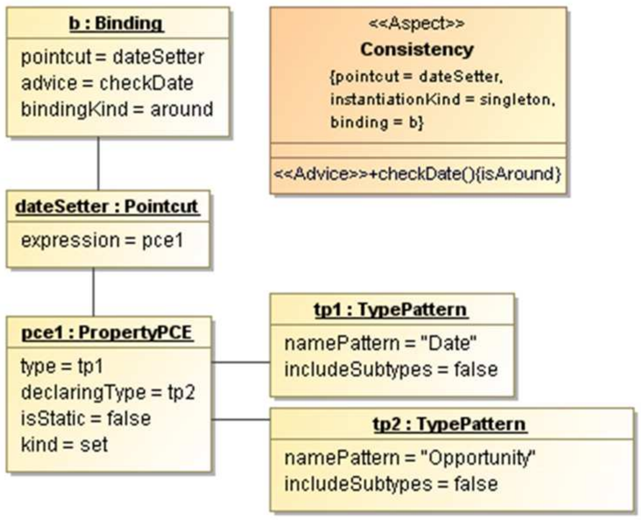

Figure 23: Pointcut and Binding

After modelling the base model, the pointcuts, and the aspects the aspect composition process starts as described in the past sections of this chapter.

The first transformation is for pointcut matching. This transformation was described in section 4.5.1. The “set new date” action of the base model (see Figure 21) matches the pointcut and is annotated with a <<JPShadow>> stereotype (see Figure 24).

During the aspect composition the weaving transformation replaces the join point stereotype by model elements which get the aspect instance, call the advice and create a closure object because the advice is woven around the join point. The closure object stores all parameters which were passed to the base join point to allow calling this base join point from within the advice operation. The resulting model is shown in Figure 25.

The aspect composition transformation also converts all aspects into object-oriented classes. All special actions are translated to plain UML/VIDE model actions. The around advice of the aspect model is replaced by an operation with a closure parameter (see section 4.5.2 for details). The "proceed" action of the advice in the aspect model is converted to a CallOperationAction which calls an operation of the closure object parameter. The result is shown in Figure 26.

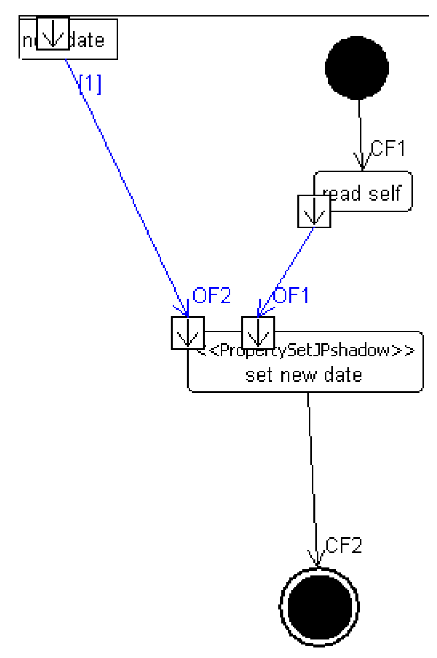

Figure 24: Joinpoint Shadow Model

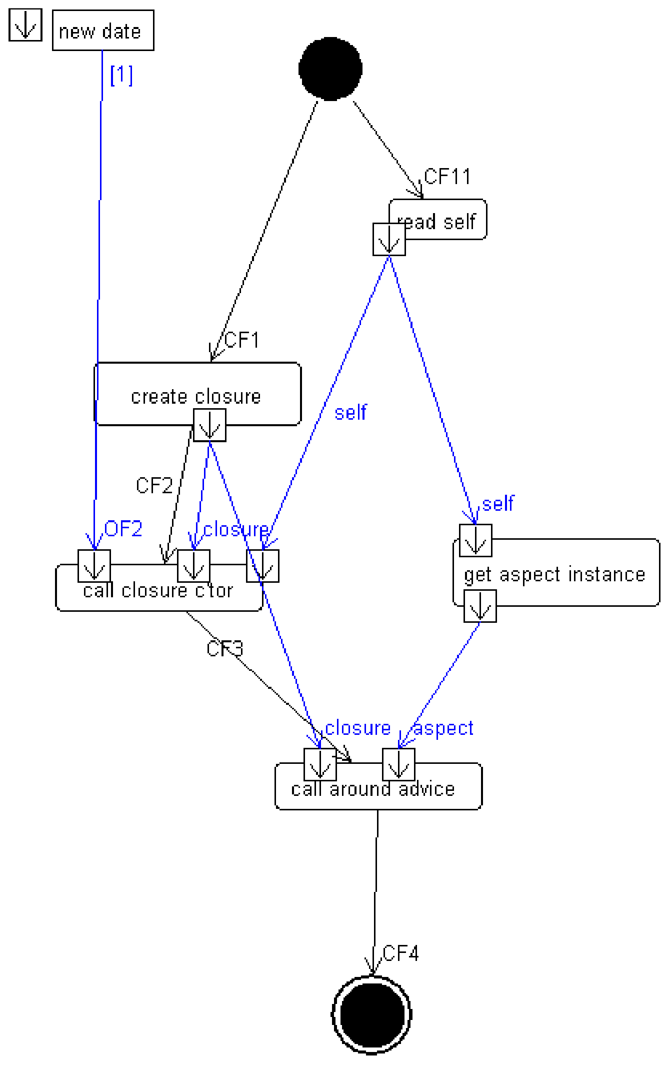

Figure 25: After weaving

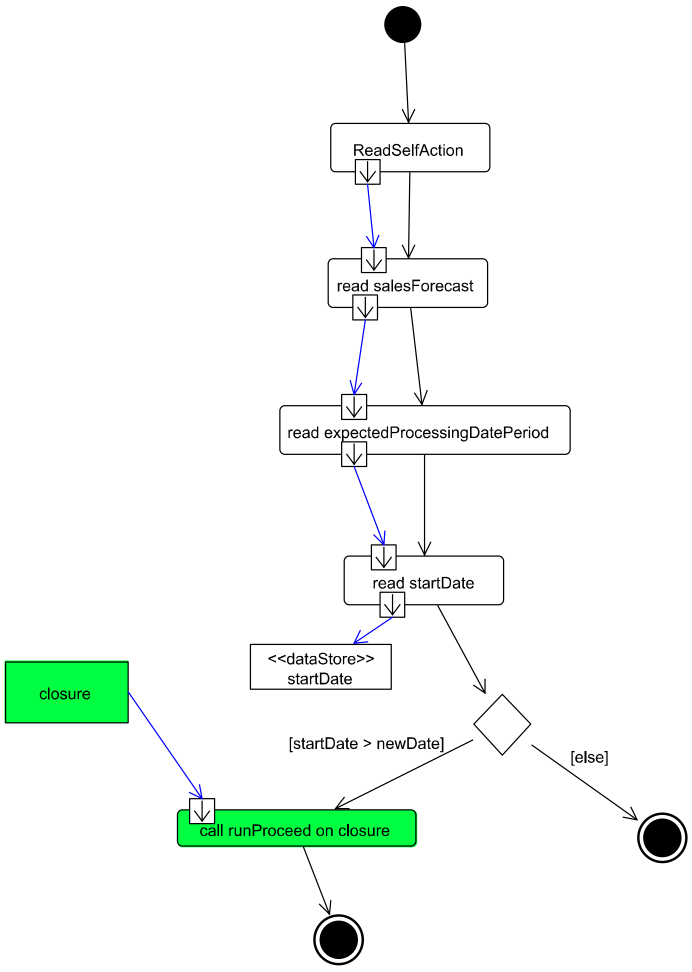

Figure 26: Advice operation after weaving

Summary

The weaving transformation replaces all join point shadows from the join point shadow model with model elements that integrate the aspect behaviour. Different adaptations are needed, depending on the join point kind (e.g. call vs. execution), binding kind (before, after, around) and aspect instantiation kind defined (e.g. singleton vs. instance). As a result all dependencies to AO profiles and models are removed. The result is a plain VIDE model to be further processed by the model compiler.

5 Summary and Conclusions

Both Aspect-Oriented Software Development (AOSD) and Model Driven Development (MDD) are approaches to reduce complexity in software development. These approaches use different but complementary ideas to reduce complexity. AOSD adds additional modules and a weaving mechanism to extract tangled and scattered functionality, so called crosscutting concerns. MDD reduces complexity by replacing the writing of source code by using abstract models instead; executable code is generated from the models. Crosscutting concerns appear already in the modelling phases of MDD while aspect-oriented programs can have a lot of source code. So it seems natural that a combination of the two approaches can have the advantages of both and thus can help overcome complexity in software development. Task 3.1 was aimed at checking whether such a concept of Aspect-Oriented Modelling (AOM) is realizable.

Chapter 3 presented an object oriented business application as an example application that had been realized using a traditional object oriented design. Such a design has its limitations especially w.r.t. the modularization of crosscutting concerns. On the basis of two examples of crosscutting concerns it was shown that the modelling of the respective behaviour using an object-oriented caused problems that are not solvable with regular MDD techniques. These examples show the need for Aspect-Oriented Modelling

A proposal for the integration of aspect-oriented concepts into MDD was given in Chapter 4. In order to develop the proposal some other concepts had to be defined after requirements for them had been explored. The first was the concept of modelling aspects on the PIM level which included an extension of the UML/VIDE metamodel. The second was a model-to-model transformation from an aspect-model and a base model into a plain UML/VIDE-model.

The model-to-model transformation was realized as a so called *Demonstrator*, a prototypic implementation in the sense of a proof-of-concept. The Demonstrator uses ATL as the transformation description language. Choosing ATL allowed us to focus on the actual transformations since the model query and modification part is implemented by the creators of the ATL implementation.

An aspect composition strategy was developed by FIRST. The aspect composition takes place completely on the PIM level resulting in a model-to-code transformation which is totally unaware of aspects. The chosen strategy requires the advice operations to be modelled using UML actions.

The composition transformation process is split into two phases: Matching phase and weaving phase. The Demonstrator implements one composition strategy, but due to the modular design that we choose it is possible to implement another strategy. In that case only the weaving transformations have to be changed. It is even possible to use a different aspect metamodel, which only requires a change of the join point matching transformations but not of the weaving transformations.

The Demonstrator was able to leverage the concept of AOM for some parts of the examples from chapter 4. It thus indicated that the AOM concept is feasible and will be able to help to circumvent the problems that arise when modelling crosscutting concerns with regular MDD.

5.1 Open issues

Although many issues had been solved some remain still open, they are presented here.

The examples presented in chapter 4 had been a good basis for gathering most of the necessary requirements. In addition to them other examples should help in identifying special requirements on the weaving strategy and the expressiveness of pointcut expressions.

A further issue is about problems that arise when weaving advices to behaviour modelled by activities. The same flow of control may be modelled in different ways, the model looks different but the modelled behaviour is the same. This fact results in the need for one weaving strategy per way of modelling a control flow. Otherwise the weaving would only lead to correct results when applied to the way of modelling the strategy was initially developed for. For this a decision has to be made either by restricting the way of modelling the flow of control or by providing respective weaving strategies.

A syntax in terms of common VIDE syntax for the pointcuts both on graphical and textual language level are not defined yet. For now pointcuts, advice and aspects can be defined only with a prototypic enhancement of standard UML.

Currently, the use of OCL instead of read actions is not supported, because of lacking tool support. The integration with OCL abstract syntax should be no problem for the model compiler, but it will not be an option for the demonstrator. However, string representations of OCL expressions are highly discouraged (for the abstract syntax level).

5.2 Outlook

Task 3.2 is about the comparison of the AOM approach and the traditional object-oriented MDD to examine the benefits and drawbacks of introducing aspects into modelling. For the purpose of that comparison some measurements will be developed. These will provide means to evaluate different viewpoints which indicate software complexity like software metrics for object-oriented code. Examples for complexity measures are the number of model elements or the number of relations between model elements. It is also possible to compare different composition strategies or different UML AO Profiles. More examples for crosscutting concerns will be provided to compare the AOM approach with OO modelling and draw conclusions supported by facts. The examples will be modelled with and without aspects and the criteria will be applied to find out whether AOM is useful for reducing complexity. Different composition strategies and AO Profiles can be used for the same example to find out which approach is the best.

References

1. SAP AG, SAP Netweaver Developer Studio, http://help.sap.com/saphelp_nw04/helpdata/en/cb/f4bc3d42f46c33e10000000a11405a/frameset.htm

2. Michael Altenhofen, Thomas Hettel, Stefan Kusterer: *OCL Support in an Industrial Environment*. MoDELS Workshops, pp.169-178, Genova, Italy, October 2006

3. Hippner, H. and Wilde, K. D. (2002). CRM—Ein Überblick, in S. Helmke, M. Uebel and W. Dangelmaier (eds), Effektives Customer Relationship Management: Instrumente—Einführungskonzepte—Organisation, second edition, Gabler, Wiesbaden, pp. 3–37.

4. Walser, K. (2002). Integrierte Prozessabwicklung aus Sicht der Kundenbeziehung—Eine Übersicht, in M. Meyer (ed.), CRM-Systeme mit EAI - Konzeption, Implementierung und Evaluation, Vieweg, Wiesbaden, pp. 61–86.

5. SAP AG, *SAP CRM*, http://www.sap.com/solutions/business-suite/crm/index.epx

6. Amberg, M. and Schumacher, J. (2002). *CRM-Systeme und Basistechnologien*, in M. Meyer (ed.), CRM-Systeme mit EAI - Konzeption, Implementierung und Evaluation, Vieweg, Wiesbaden, pp. 21–59.

7. Hippner, H., Hoffmann, O., Rimmelspacher, U. and Wilde, K. D. (2006). *IT Unterstützung durch CRM-Systeme am Beispiel von mySAP CRM*, in H. Hippner and K. D. Wilde (eds), Grundlagen des CRM, second edn, Gabler, Wiesbaden, pp. 15–44.

8. TopCased, http://www.topcased.org/

9. Khanna, A. *How to set up partner determination in mySAP CRM*, CRM Expert http://www.crmexpertonline.com/archive/Volume_03_(2007)/Issue_01_(January_and_February)/v3i1a3.cfm

10. SAP AG, Partner Determination Procedures, SAP Library http://help.sap.com/saphelp_crm40/helpdata/en/3c/92ecee484a11d5980800a0c9306667/content.htm

11. Filman R, Friedman P., Aspect-Oriented Programming is Quantification and Obliviousness, technical report, 2001 http://www.riacs.edu/research/technical_reports/TR_pdf/TR_01.12.pdf

12. Buck-Emden R., Zencke, P., mySAP *CRM: The Offcial Guidebook to SAP CRM Release 4.0*, SAP Press, May 2004

13. Aspect-oriented Model-driven Product Line Engineering (AMPLE), http://www.ample-project.net/

14. Feature-getriebene, aspektorientierte und modellgetriebene Produktlinienentwicklung, http://www.feasiple.de/

15. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12) (1972) 1053-1058

16. Aspectj homepage, October 2006. http://www.eclipse.org/aspectj/.

17. Yan Han, Günter Kniesel, Armin Cremers: Towards Visual AspectJ by a Meta Model and Modelling Notation, Proceedings of the 6th International Workshop on Aspect-Oriented Modelling, Chicago, USA, March 2005

18. Christina von Flach G. Chavez & Carlos J. P. Lucena A Theory of Aspects for Aspect-oriented Software Development. 1st Brazilian Symposium on Software Engineering, pp. 130-145, Manaus, Brazil, 2003

19. Siobhán Clarke, Robert Walker: Towards a Standard Design Language for AOSD, Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD 2002), pp. 113-119, Enschede, The Netherlands, March 2002

20. Andrea Schauerhuber, Wieland Schwinger, Elisabeth Kapsammer, Werner Retschitzegger, Manuel Wimmer: Towards a Common Reference Architecture for Aspect-Oriented Modelling, Proceedings of Workshop on Aspect-Oriented Modelling, Fifth International Conference on Aspect-Oriented Software Development, Bonn, Germany, March 20-24, 2006

21. Lidia Fuentes, Pablo Sánchez: Elaborating UML 2.0 Profiles for AO Design, Proceedings of Workshop on Aspect-Oriented Modelling, Fifth International Conference on Aspect-Oriented Software Development, Bonn, Germany, March 20-24, 2006

22. Gefei Zhang: Towards Aspect-Oriented Class Diagrams, Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), pp. 763-768, Taipei, Taiwan, December 15-17

23. Awais Rashid, Alessandro Garcia, Ana Moreira: Aspect-Oriented Software Development Beyond Programming, Proceedings of 9th International Conference on Software Reuse, pp. 441-442, Torino, Italy, June 11-15, 2006

24. Dominik Stein, Stefan Hanenberg, Rainer Unland: Position Paper on Aspect-Oriented Modelling: Issues on Representing Crosscutting Features, Proceedings of Third International Workshop on Aspect Oriented Modelling, Boston, USA, March 17-21, 2003

25. Jackson, Andrew and Clarke, Siobhán. Initial Version of Aspect-Oriented Design Approach. Trinity College Dublin, AOSD-Europe Deliverable D38, AOSD-Europe-TCD-7, February 2006

26. Iris Groher, Stefan Schulze: Generating Aspect Code from UML Models, Proceedings of Third International Workshop on Aspect Oriented Modelling, Boston, USA, March 17-21, 2003

27. Chitchyan, Ruzanna and Rashid, Awais and Sawyer, Pete and Garcia, Alessandro and Alarcon, Mónica Pinto and Bakker, Jethro and Tekinerdogan, Bedir and Clarke, Siobhán and Jackson, Andrew. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Lancaster University, AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, May 2005

28. D. Wampler: The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture, 2003

29. B. Baudry et al.: Exploring the Relationship between Model Composition and Model Transformation, in AOM-WS at MoDELS 2005

30. B. Tekinerdogan, M. Aksit, F. Henninger: Impact of Evolution of Concerns in the Model-Driven Architecture Design Approach, in ABMB at ECMDA-FA 2006

31. D. Simmonds, A. Solberg, R. Reddy, R. France, S. Ghosh: An Aspect Oriented Model Driven Framework, Proceedings of the 9th International Enterprise Distributed Object Computing Conference (EDOC 2005), IEEE Computer Society Press, pp. 119-130, Enschede, The Netherlands, September 19-23, 2005

32. S. Clarke and E. Baniassad: Aspect-Oriented Analysis and Design - The Theme Approach, Addison-Wesley, 2005

33. P. Amaya, C. Gonzalez, J.M. Murillo: Towards a Subject-Oriented Model-Driven Framework, in ABMB at ECMDA-FA, 2005

34. Straw et. al.: Model Composition Directives, Proceedings of the 7th UML Conference, Lisbon, Portugal, October 10-15, 2004

35. NoE AOSD-Europe report D9 IST-2-004349-NOE AOSD-Europe

36. Filman, R., et al., Aspect-Oriented Software Development. 2004: Addison-Wesley.

37. R. Laddad, AspectJ in Action, 2003, Manning Publications

38. E. W. Dijkstra, On the role of scientific thought (published as EWD447), Aug 1974, http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF

39. K. van den Berg, J.M. Conejero, R. Chitchyan, AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation, 2005, Technical Report, AOSD Europe

40. Ch. Koppen, M. Stoerzer, PCDiff: Attacking the Fragile Pointcut Problem, In: Proceedings on 1st European Interactive Workshop on Aspects in Software (EIWAS), 2004

41. E. Hilsdale, J. Hugunin, Advice Weaving in AspectJ, Mar 2004, AOSD04

42. ATLAS Transformation Language (ATL), http://www.eclipse.org/m2m/atl/

43. M. Eichberg: MDA and Programming Languages, In Proceedings of the Workshop on Generative Techniques in the context of Model Driven Architecture. OOPSLA, November 2002

44. V. Kulkarni and S. Reddy: Supporting Aspects in MDA, Proc. of the Workshop in Software Model Engineering on the UML'2003, San Francisco, USA, 2003

45. D. Wampler: The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture, 2003

46. D. Stein, S. Hanenberg, and R. Unland: Modeling Pointcuts, Proc. of the 7th International Conference on the Unified Modeling Language (UML 2004), Lisbon, Portugal, October 11-15, 2004, Springer, LNCS 3273, pp. 98-112

47. P.A. Amaya Barbosa , C.F. Gonzalez Contreras, J.M. Murillo Rodriguez: MDA and Separation of Aspects: An approach based on multiple views and Subject Oriented Design, Proc. of 5rd International Workshop on Aspect-Oriented Modeling with UML, AOSD 2005, Chicago, IL, 2005

48. M. Mezini and K. Ostermann: A Comparison of Program Generation with Aspect-Oriented Programming, In Proc. of the EU-NSF Strategic Research Workshop on Unconventional Programming Paradigms. Springer Verlag LNCS 3566

49. Omar Aldawud: A UML Profile for Aspect Oriented Programming, Workshop on Aspect-Oriented Programming in conjunction with OOPSLA . Tampa, Florida, 2001

DISCLAIMER OF SAP AG